RTOS '88

The Fifth Workshop on Real-Time Software and Operating Systems

May 12, 13, 1988 • Washington, D.C.

Sponsored by IEEE Computer Society and USENIX Association

Abstracts of IEEE and USENIX of

the Fifth Workshop on

Real-Time Software and Operating Systems

12 - 13 May, 1988

Washington, D.C.

General Chairman:
John A. Stankovic, University of Massachusetts

Program Committee:

Marc Donner, IBM Research
Lui Sha, Carnegie Mellon University

Technical Committee:

Marc Donner, IBM Research
Michael Hawley, Massachusetts Institute of Technology
Douglass Locke, IBM Federal Systems Division
Lui Sha, Carnegie Mellon University

sponsored by
IEEE Computer Society and USENIX Association

Preface

Welcome to the Fifth Workshop on Real-Time Software and Operating Systems. This year, for the first time, the workshop is a joint effort of the IEEE Technical Committee on Real-Time Systems and of the Usenix Association.

A workshop differs from a conference in that the former focuses on the exchange of ideas rather than the presentation of results. In a workshop, presenters should be discussing problems and techniques - with a particular emphasis on work-in-progress and unresolved difficulties. The program committee has taken this notion to heart in the selection of submissions for presentation and in the panel session. Our hope is that you, the participants, will find the result technically stimulating and that you will participate vigorously in discussion and even argument.

Finally, we would like to acknowledge the contributions of the many people without whom this workshop would never have come about. Joan Maddamma of CMU and Judy DesHarnais of Usenix did most of the real work, and Doug Locke of IBM FSD and Mike Hawley of the MIT Media Lab and NeXT Inc served on the technical committee, a thankless but critical job, and finally Peter Salus of Usenix proposed and supported this joint effort.

Jack Stankovic, General Chairman
University of Massachusetts

Marc Donner, Program Co-Chairman
IBM Research

Lui Sha, Program Co-Chairman
Carnegie Mellon University

Presentation Selection Procedure

In choosing papers for presentation at the workshop, the program committee used a two phase procedure. First, each of the four members read the 76 abstractions and ranked them according to the criteria listed below. Second, 20 papers were selected by the consensus of the program committee. Papers were graded loosely in three broad areas:

1. Systems -- Was it a design proposal or was something tangible really built?
2. Theory -- Did the authors prove a major result?
3. Ideas -- Did it present an exciting original idea that researchers and practitioners should know?

To provide a balanced program, we scheduled four pannels in addition to the presentations and we tried to ensure the presentation covers each of the following topic areas.

1. Real-Time Scheduling Techniques
2. Languages and OS
3. Real-Time Operating Systems
4. Real-Time Applications
5. Real-Time Databases

In retrospect, two aspects seemed especially important: novelty in approach and challenging real applications.

PROGRAM COMMITTEE:

Marc Donner, IBM Research
Michael Hawley, Massachusetts Institute of Technology
Douglass Locke, IBM Federal Systems
Lui Sha, Carnegie Mellon University

TABLE OF CONTENTS

Panel: Building Real-Time Kernels
Chairman: A. M. van Tilborg, Office of Naval Research.
Panelists: A. K. Agrawala, University of Maryland/ R. Cook, University of Virginia/
A. Shaw, University of Washington/ K. Shin, University of Michigan/
H. Tokuda, Carnegie Mellon University/ H. F. Wedde, Wayne State University

SESSION I

REAL-TIME SCHEDULING TECHNIQUES

Chairman

Jane W. S. Liu
University of Illinois at Urbana-Champaign

Scheduler 1-2-3 : It's better to be predictable than *ad hoc*.

Hideyuki Tokuda and Makoto Kotera

Computer Science Department
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
(412) 268-7672

1. Introduction to the tool set for Real-Time Systems

Advances in Software Engineering provided us a set of modern programming tools for building a large complex software system. In non real-time system development, we may enjoy chasing bugs with those tools throughout the software development cycle from a requirement analysis phase to debugging and maintenance phases. However, lack of timing capability makes them ineffective to build or analyze complex real-time systems. Timing bugs are really hard to be found without effective software tools. They are running around beyond the task boundary and confuse the designers to track down their real origins. An additional tool set such as a timing tool, a schedulability analyzer and a real-time monitor/debugger should be developed to capture timing bugs at the point where they originates.

In this paper we describe a interactive tool for the schedulability analysis, called *Scheduler 1-2-3*[1]. By the schedulability analysis, we mean the analysis which verifies whether or not all "hard" real-time task activities will complete by its "hard" deadline time. There are well-known schedulability analysis algorithms [4, 5, 7], however, practical interactive analysis tools have not been demonstrated yet. *Scheduler 1-2-3* is a window-based interactive tool. It can be also used as a synthetic workload generator as a part of an integrated tool set that consists of a timing tool, a schedulability analyzer and a real-time monitor/debugger. Before describing *Scheduler 1-2-3*, let us sketch out other related approaches to the schedulability analysis.

2. Schedulability analysis

Schedulability analysis of hard deadline tasks is hard to perform. However, because of its importance, several analysis approaches have been already proposed. We review four approaches of them. Those are the rate monotonic algorithm [4], the Reinbaugh's formula [7], the approach based on Real-Time Euclid programming language [2, 9] and the Real-Time Temporal Logic approach by Ostroff [6].

The rate monotonic algorithm brought us a well-defined theoretical result of schedulability analysis. The rate monotonic scheduling gives higher priority in execution to more frequently executed periodic tasks, while priorities are given in *ad hoc* manner by most existing systems. Under the rate monotonic scheduling algorithm, if processor (CPU) utilization is less than $n \cdot (2^{1/n} - 1)$, which implies $ln\, 2 \simeq 0.693$, with n periodic tasks in the given task set, it has been proved to be schedulable.

A well-known formula for the schedulability analysis was given by Reinbaugh. This approach can be applied to arbitrary real-time systems because of its scheduling policy independent approach, however, the analysis is rather pessimistic. In his approach, an execution time of a hard deadline task consists of critical sections, non-critical sections and blockage delays. Assuming that critical sections and blockage delays are not preemptable, schedulability is checked by means of the total processor time ratio allocated to each task to perform computation of non-critical sections. The processor ratio for each task is

---

[1]Of course, the name of *Scheduler 1-2-3* came from *Lotus 1-2-3* which is a registered trade mark of Lotus Development Corporation.

depending on scheduling policy that the target system employs.

The approach based on Real-Time Euclid uses a set of schedulability provisions built-in for schedulability analysis. This approach takes advantage of explicit bindings between program code segment and timing constraints. In traditional real-time systems, those bindings were implicit. On the other hand, the applicable range is limited due to its language oriented approach. The analysis tool, called *Schedalyzer*, consists of the front end and back end. The front end collects timing information on the target system. The back end utilizes the Reinbaugh's analysis algorithm with the earliest deadline first scheduling.

Another approach which utilizes Real-Time Temporal Logic (RTL) and Extended State Machine (ESM) is proposed in recent. The analysis technique based on Real-Time Temporal Logic could provide us a scheme to verify *timing correctness* by giving timing specification of the target system with RTL. It is much hard to verify the *timing correctness* of the target systems with traditional approach, even though it is possible to track down timing errors in real-time systems due to violations of *timing correctness*.

3. *Scheduler 1-2-3*

The main objective is to provide an interactive design tool that makes it easy to design a complex real-time system. For the existing systems, the *Scheduler 1-2-3* can be used to predict the timing effects due to the software and hardware modification. Another objective is to utilize *Scheduler 1-2-3* as a synthetic workload generator, which can be integrated with the other test tools, the timing tool and the real-time monitor/debugger. The objectives are summarized the following.

- **Schedulability analysis**

  Schedulability is verified for the given hard deadline task sets under arbitrary scheduling algorithms such as the rate monotonic, the first-in first-out (FIFO), the earliest deadline first and so on. If a given task set is not schedulable, the time when a deadline will be missed first and the task which miss a deadline are given to the user; furthermore, some intelligent suggestions are also given. For instance, the way how a nonschedulable task set can become schedulable are suggested.

- **Response time analysis for aperiodic tasks**

  Performance of the given sets of aperiodic tasks with soft deadlines is given under arbitrary scheduling algorithms for aperiodic tasks. By means of a simulation, *Scheduler 1-2-3* estimates the average response times of those aperiodic tasks based on the given statistical parameters, like its mean arrival time, standard deviation and distribution.

- **Monitorability analysis**

  Time critical nature of real-time systems necessitates to minimize and preestimate the interference of the runtime monotor/debuggers. Our runtime monitoring approach [10] adds a monotoring task in advance to the target task set so that monitoring overhead can be analyzed by *Scheduler 1-2-3* . In addition, *Scheduler 1-2-3* predicts the maximum capacity of the monitoring task under a given scheduling policy[2] so as not to have the monitoring capacity overwhelmed by too frequent event occurrence.

- **Easy-to-use interface**

  An interactive user interface is provided on a window system for ease of use. This interface makes it possible for users to enjoy schedulability analysis without precise knowledge of internal analysis procedures.

- **Synthetic workload generator**

---

[2]For the sake of simplicity, we call this prediction *monitorability analysis*.

To confirm the schedulability of the given task set in a practical environment, *Scheduler 1-2-3* outputs a workload table on which a synthetic real-time task set can be generated. Synthetic tasks with workload specified would be tested in the ART Real-Time testbed.

Scheduler 1-2-3 consists of three major modules. They are the Analysis module, the File interface module and the User interface module, as depicted in Figure-1. It is easy to port *Scheduler 1-2-3* to other hosts or to apply the analysis module to another purpose thanks to the independent structure of the major modules.

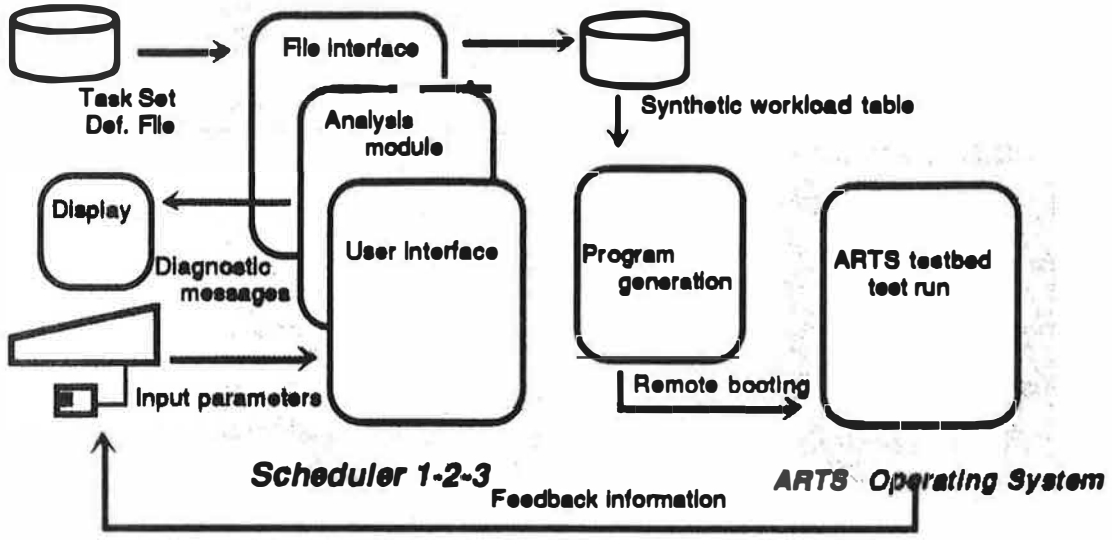


Figure-1 System flow in the ART Real-Time testbed: The synthetic workload table is included into the test program. The test program with workload specified is tested on the ARTS kernel. Feedback information will be given to the user through the ART Runtime Monitor [10].

- **Analysis Module**

  The analysis module performs schedulability analysis, response time analysis for aperiodic tasks and monitorability analysis. The rate monotonic based algorithm [8] is currently available for schedulability analysis of hard deadline tasks. The scheduling algorithm for aperiodic tasks is currently based on the deferrable server algorithm [3].

- **File Interface Module**

  Generating a synthetic workload table and reading/writing a task definition file are done by this module. As depicted in Figure-1, a test program is created from a synthetic workload table to be executed on the ART Real-Time testbed. A task set definition table is a collection of binary data of the parameters of the given task set. This table is used to save the time to re-input the task set definition.

- **User Interface Module**

  This module provides an easy-to-use interactive user interface based on a window system. The current implementation has been done on the window system of Sun3 workstation[3]. It is possible both to use *Scheduler 1-2-3* on windowless hosts and to port it on other window systems like X-window system.

---

[3]Sun3 is the trademark of Sun Microsystems, Inc.

Operations are triggered by clicking button images on the top subwindow with a mouse. An analysis is performed for the Inertial Navigation System task set [1] in Figure-2. All necessary task definitions were already set through the bottom subwindow. By clicking the button image, Cyclic, the analysis is started, and then the result of analysis shows this task set is schedulable with 0.86 CPU utilization as displayed in the small box. Meanwhile, Figure-3 indicates a nonschedulable case due to the context switching overhead.

Figure-4 demonstrates another facility which makes *Scheduler 1-2-3* an efficient integrated tool. For the purpose of testing in the ART Real-Time testbed, *Scheduler 1-2-3* generates workload tables after the schedulability analysis. The workload table would be included into testing programs that put synthetically generated workload, then schedulability is practically tested in the ART Real-Time testbed. A workload table for the INS task set is shown in Figure-4, which would be an include file of C programs.

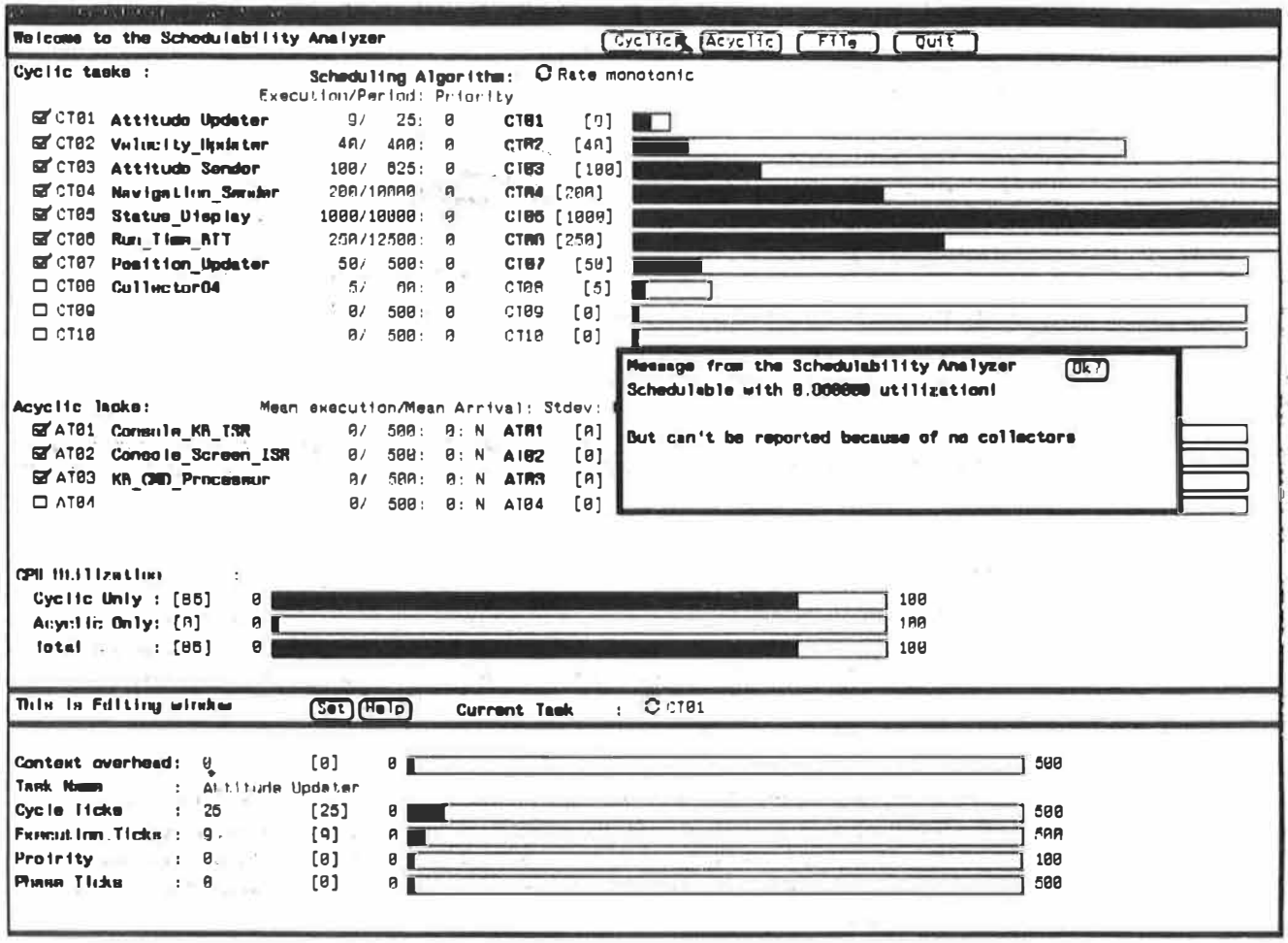


Figure-2 The INS task set: The schedulability of the INS task set is being verified. *Scheduler 1-2-3* reports this task set is schedulable with 0.86 CPU utilization. Alas, monitoring the system behavior is impossible without a monitoring task.

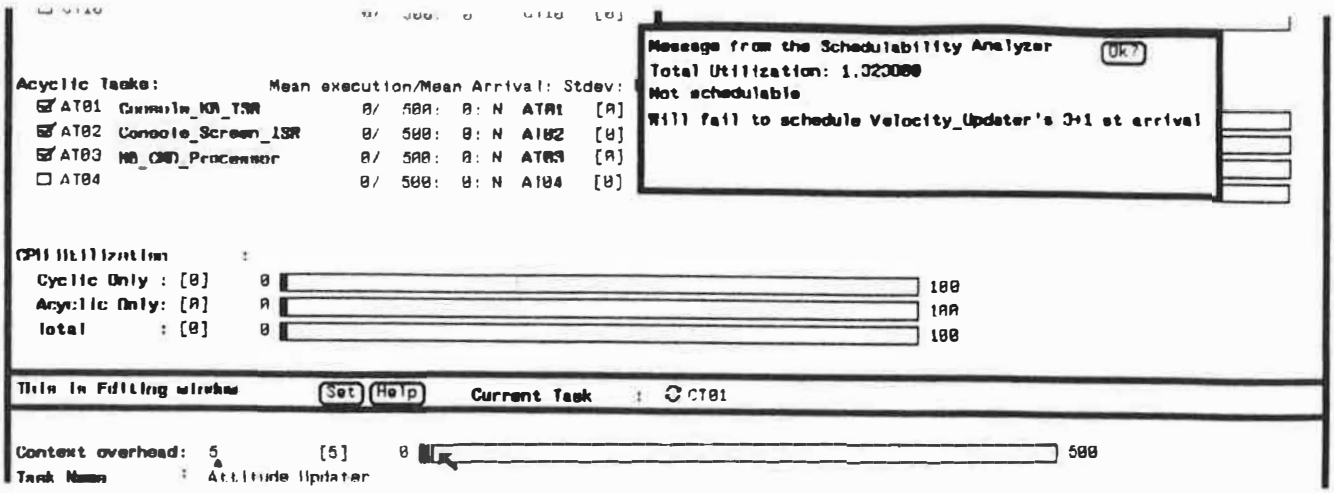


Figure-3 The effect of context switching overhead: The task set that used be schedulable in Figure-2 becomes nonschedulable by adding five ticks as the context switching overhead. Ticks represents time units on the target system.

```
typedef       struct{
      char    t_avail;
      char    t_name[MAXTASKNAME];
      char    t_type;
      unsigned int    t_period;
      unsigned int    t_exec;
      unsigned int    t_prio;
      unsigned int    t_phase;
} XL_SCHED_ELM;
```

Data structure

```
'\001', "Attitude_Updater", '\000', 25, 9, 0, 0,
'\001', "Velocity_Updater", '\000', 400, 40, 0, 0,
'\001', "Attitude_Sender", '\000', 625, 100, 0, 0,
'\001', "Navigation_Sender", '\000', 10000, 200, 0, 0,
'\001', "Status_Display", '\000', 10000, 1000, 0, 0,
'\001', "Run_Time_BIT", '\000', 12500, 250, 0, 0,
'\001', "Position_Updater", '\000', 500, 50, 0, 0,
'\000', "NA", '\000', 500, 0, 0, 0,
'\000', "NA", '\000', 500, 0, 0, 0,
'\000', "NA", '\000', 500, 0, 0, 0,
'\001', "Console_KB_ISR", '\001', 500, 20, 0, 0,
'\001', "Console_Screen_ISR", '\001', 500, 54, 0, 0,
'\001', "KB_CMD_Processor", '\001', 500, 0, 0, 0,
'\000', "NA", '\001', 500, 0, 0, 0,
```

File format

Figure-4:The workload table for the INS task set: An array of the data structure, XL_SCHED_ELM, is used to include this workload file into C programs. The C programs generated workload specified by the table on ARTS kernel.

4. Summary

Despite advances of Software Engineering, there has not been effective tools to capture timing bugs. On the other hand, it is indispensable for real-time systems to capture timing bugs at the point where they come. In this paper, we introduced an interactive schedulability analyzer, *Scheduler 1-2-3* which can analyze whether a given task set can meet their timing constraints or not at the system design phase. This tool gives us some information on the habitat of timing bugs. The ease of use and practicality of

Scheduler 1-2-3 is illustrated by a few examples of actual operations on the INS task set.

An important extension is to provide analysis capabilities under other scheduling policies other than the rate monotonic. Response time analysis for aperiodic tasks is another immediate extension. This analysis should be performed not only by the deferrable server algorithm but also by background, polling, priority exchange algorithms. Diagnostic messages given by *Scheduler 1-2-3* should be more sophisticated. For instance, intelligent messages (e.g. a result from a period transformation method) to improve the schedulability should be included. Reversed analysis is also in demand. That is, for a given response time requirement of the aperiodic task set, *Scheduler 1-2-3* should be able to estimate the necessary specifications of the deferrable server or anything that can produce the given response time. The goal of this reversed analysis would be to exploit a scheduling scheme for a mixture of periodic and sporadic tasks.

Reference

[1] Borger, M. W.
VAXLEN Experimentation: Programming a Real-Time Perodic Task Dispatcher using VAXLEN Ada 1.1.
Technical Report CMU/SEI-87-TR-32(ESD-TR-87-195), Carnegie Mellon University, September, 1987.

[2] Kligerman, E. and Stoyenko, A. D.
Real-Time Euclid: A Language for Reliable Real-Time Systems,
IEEE Transaction on Software Engineering , September, 1986.

[3] Lehoczky, J. P., Sha, L. and Strosnider, J. K.
Enhanced Aperiodic Responsiveness in A Hard Real-Time Environments.
In *Proceedings of 8th IEEE Real-Time Systems Symposium.* December, 1987.

[4] Liu, C. L. and Layland, J. W.
Scheduling Algorithm for Multiprogramming in a Hard Real-Time Environment.
JACM 20(1), 1973.

[5] Mok, A. K.
Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.
PhD thesis, Massachesetts Institute of Technology, May, 1983.

[6] Ostroff, J. S. and Wonham, W. M.
Modelling, Specifying and Verifying Real-Time Embedded Computing Systems.
In *Proceedings of 8th IEEE Real-Time Systems Symposium.* December, 1987.

[7] Reinbaugh, D. W.
Guaranteed Response Times in a Hard-Real-Time Environment.
IEEE Transaction on Software Engineering , January, 1980.

[8] Sha, L., Lechoczky, J. P. and Rajkumar, R.
A Schedulability Test For Rate-Monotonic Priority Assignment.
Computer Science Department ART project, Carnegie Mellon University.
July, 1987

[9] Stoyenko, A. D.
A Schedulability Analyzer for Real-Time Euclid.
In *Proceedings of 8th IEEE Real-Time Systems Symposium.* December, 1987.

[10] Tokuda, H., Kotera, M. and Mercer, C. W.
A Real-Time Monitor for a Distributed Real-Time Operating System.
In *Proceedings of SIGOPS and SIGPLAN workshop on parallel and distributed debugging.* May, 1988.

Specifying Scheduling Paradigms for Time Dependent Processes *

I. Lee, R. Gerber and A. Zwarico
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

March 1, 1988

1 Introduction

Recently there has been a spate of effort directed toward the modeling of real-time programs. Mok's group at Texas [1] has developed Real-Time Logic (RTL), which expresses a time-dependent system in terms of first order logic. De Roever's group at Eindhoven [2] has defined a complex algebra which essentially captures the real-time aspects of Ada. At the University of Pennsylvania we have developed the timed trace model [3], which is a time-based extension of Hoare's trace algebra [4].

All of these models are striving toward a similar goal: that one should be able to abstractly specify a system's timing constraints, and then prove (or disprove) that they are mutually consistent.

Under-Specification. The complexity of the specification-verification relationship is aggravated by a fundamental problem – these two functions possess requirements which are mutually incompatible. By definition, a specification must capture *at least* all of the behaviors of its target system. But because the specification should be an abstract and concise statement of timing constraints, it typically under-specifies the system. Thus, the derived set of potential behaviors is artificially inflated, and verification becomes all the more difficult. Also, it often happens that the specification captures behaviors with "inconsistent" timing constraints, whereas the underlying system precludes such behaviors.

Scheduling. While the problem of under-specification cannot be eliminated, one of its primary causes should be addressed. A system's process scheduler typically disallows most of the specified system behaviors. Thus, we argue that the scheduling function should not be ignored in the specification – to do so makes the verification function unnecessarily complex.

Two approaches have been taken here, both of which are extreme. The first is to express the scheduling function within the specification itself; we refer to this technique as an *explicit schedule*. As we shall show, formulating an explicit schedule is at times possible. However, it typically results in a specification whose complexity exceeds that of the complete system realization in a programming language. Thus, this method compromises the main objective of a specification, that of simplicity.

*This research was supported in part by NSF DCR 8501482, NSF DMC 8512838, NSF MCS 8219196-CER, ARO DAA6-29-84-k-0061, and a grant from AT&T's Telecommunications Program at the University of Pennsylvania.

The alternative technique is to use an *implicit schedule*. This method incorporates a scheduling discipline into the formal model itself – some notion of a schedule is fundamental to the model's operational semantics. As an example, Koymans *et al.* [2] employs the semantics of "maximal parallelism" that were defined, in an untimed setting, by Salwicki and Müldner [5]. The model assumes that a process action is never unnecessarily delayed in a "ready" state. While this model possesses desirable algebraic properties, it also imposes several unrealistic constraints on a specification. For example, it assumes a one-to-one correspondence between processes and processors.

In this paper we briefly describe the timed trace model. We then present two representative examples which display the need for scheduling disciplines. One contains an explicit scheduler within its specification. The other demonstrates a palpable requirement for implicit scheduling. We conclude with our current research direction.

2 The Timed Trace Model

In this section we briefly present the *Timed Trace* model. For a complete description of the model see [3]. The timed trace model is a temporal extension of Hoare's trace model for CSP [4] that associates an explicit occurrence time with each event that a process executes. An *event* is an instantaneous visible action in which a process engages during its execution. Time is modeled by the set, $\mathbf{N}^\infty$, containing the natural numbers $\mathbf{N}$ and ∞.

A *timed trace* is a finite sequence $\langle(a_1,n_1),(a_2,n_2),\ldots,(a_m,n_m)\rangle \in (\Sigma \times \mathbf{N}^\infty)^*$, recording the events that a process has executed up to some moment in time. Each (a_i,n_i) represents the occurrence of the i^{th} event in the execution of the process. The time n_1 is the time between 0 and the occurrence of a_1. For $i > 1$, n_i represents the relative time between events a_{i-1} and a_i. Traces are denoted by s, t and u. The empty trace is denoted by $\langle\rangle$. The concatenation of two traces s and t is written $s^\frown t$. The restriction of a trace s to a set of events A, $s \upharpoonright A$, removes all events not in A from s, while preserving the occurrence times of the events of s in A. A trace t is a prefix of s, $t \leq_{tr} s$, if and only if there is some trace u such that $t^\frown u = s$. Every trace has $\langle\rangle$ as a prefix. Incrementing a trace t by i, $t + i$, increments the occurrence time of the first event in t by i. If $t = \langle\rangle$ then the operation has no effect.

Mathematically, a deterministic real-time process P is a pair $(\bar{\alpha}P, TRS(P))$ where $\bar{\alpha}P$ is the *alphabet* of P and $TRS(P)$ is its *trace set*. The alphabet is the set of events P can execute. The trace set represents all possible executions of P. For example, $\mathrm{STOP_A} = (\mathrm{A}, \{\langle\rangle\})$ is a process with alphabet A that does nothing, and $(\{a,b\}, \{\langle\rangle, \langle(a,1)\rangle, \langle(b,3)\rangle\})$ is a process that either executes a at time 1 or b at time 3.

We denote the domain of deterministic real-time processes $\mathcal{TD}$. The domain is partially ordered by *process containment*. We say that P is contained in Q, $P \sqsubseteq Q$, if $\bar{\alpha}P = \bar{\alpha}Q$ and $TRS(P) \subseteq TRS(Q)$.

We define a set of operators, timed action, choice, parallelism and a recursive construct on the domain of real-time processes.

Timed action represents the sequential execution of events with respect to time. The process $a \overset{i}{\rightsquigarrow} P$ engages in event a and, after delaying for exactly i time units, behaves like the process P. In order to model a process that delays before engaging in its first event, we introduce a special symbol ϵ that marks a point in time. The process $\epsilon \overset{i}{\rightsquigarrow} P$ executes P after a delay i. The symbol

ϵ is not an event and never occurs in any trace. Thus,

$$a \overset{i}{\rightsquigarrow} P = \begin{cases} (\{a\} \cup \bar{\alpha}P, \{\langle\rangle\} \cup \{\langle(a,0)\rangle \hat{} s | s \in TRS(P) + i\}) & a \neq \epsilon \\ (\bar{\alpha}P, TRS(P) + i) & a = \epsilon \end{cases}$$

where $TRS(P) + i$ is the natural extension of addition on traces to sets of traces. To improve expressibility, we allow intervals on the timed action operator $\epsilon \overset{int}{\rightsquigarrow} P$. This process can start at any time during the interval *int*.

The choice construct, $P \,\square\, Q$, represents the deterministic choice between two processes. $P \| Q$ represents the simultaneous execution of and interaction between P and Q. Simultaneous execution is represented by interleaving the executions of P and Q in a way that preserves the occurrence times of the events in the two processes. P and Q interact only if they are able to engage in the same event simultaneously. Thus, $P \| Q$ contains all traces that belong to P and Q when they are restricted to the alphabet of P and Q respectively. Repetition is modeled using the recursive construct $\mu P.F(P)$ where F is a function composed of the above operators and P is a process identifier.

3 A Timed Producer–Consumer Problem

In this section we present an example of a problem that can be specified with an explicit schedule. Here we have two processes that may be considered "producers." Producer$_1$ generates a data item in one second (or time unit), and then may wait up to two seconds for the data to be accepted. Producer$_2$ also requires one second to produce a data item, but will only wait one additional second for this item to be accepted. Both producers execute *ad infinitum.* These processes are expressed as follows:

$$P_1 = \epsilon \overset{[1,3]}{\rightsquigarrow} a \overset{0}{\rightsquigarrow} P_1 \qquad\qquad P_2 = \epsilon \overset{[1,2]}{\rightsquigarrow} b \overset{0}{\rightsquigarrow} P_2$$

The function of the consumer C is to accept data from both producers. Thus the system is expressed by the parallel composition of the two producers and the consumer, $C \| P_1 \| P_2$. Correctness is ensured if none of the component processes ever stops. In effect, the consumer is a scheduler for the two producers. It prunes the trace set of $C \| P_1 \| P_2$ so that no trace may end in deadlock.

In this example we present a well-behaved consumer that enforces a strict round-robin schedule. Specifically, it alternates accepting data from the producers, and ensures timing consistency by processing the data in one second. It is represented by the following process:

$$C = (\mu C_1.(\epsilon \overset{1}{\rightsquigarrow} a \overset{1}{\rightsquigarrow} b \overset{0}{\rightsquigarrow} C_1)) \,\square\, (\mu C_2.(\epsilon \overset{1}{\rightsquigarrow} b \overset{1}{\rightsquigarrow} a \overset{0}{\rightsquigarrow} C_2))$$

To show that the scheduling paradigm enforced by the Consumer maintains correct execution, we find the trace set of $C \| P_1 \| P_2$ using the semantics of the operators,

$$TRS(C \| P_1 \| P_2) = \{\langle(a,1)\rangle, \langle(b,1)\rangle\} \cup \{s | s \in (\langle(a,1),(b,1)\rangle)^* \vee s \in (\langle(b,1),(a,1)\rangle)^*\}.$$

$C \| P_1 \| P_2$ does not deadlock because for all $s \in TRS(C \| P_1 \| P_2)$, either $s \hat{} \langle(a,1)\rangle \in TRS(C \| P_1 \| P_2)$ or $s \hat{} \langle(b,1)\rangle \in TRS(C \| P_1 \| P_2)$. Furthermore, we see that neither producer stops, because for all $s \in TRS(C \| P_1 \| P_2)$ there is no trace $t \in TRS(P_1)$ or $t \in TRS(P_2)$, where the length of t is greater than 1, such that $s \hat{} t \in TRS(C \| P_1 \| P_2)$.

Unfortunately, most systems do not lend themselves to snug schedules like the consumer in this example. Here we are able to satisfy the producers' requirements with a simple lock-step schedule, easily expressible in the timed trace model. This cannot be done in general, as we demonstrate in the next example.

4 Timed Dining Philosophers

We now examine a more challenging system, a timed version of the dining philosophers problem. Here the waiting and eating times for the philosophers are bounded, constraints which yield typical real-time behavior. As in [4] we consider the case of five philosophers.

Using this system we demonstrate our principal thesis: That unless the formal model includes some notion of scheduling, the system specification may become very difficult to formulate. We first postulate the problem using the weak scheduling discipline of our model's parallel operator. We show that its inherent semantics causes the system to exhibit many traces that end in deadlocked states. We further show that using the same specification, an implicit scheduling paradigm eliminates such traces.

The dining philosophers system is represented by the concurrent execution of eleven processes – five for the philosophers, five for the forks, and a footman process. The fork processes are essentially synchronizers – they ensure that each fork is held by at most one philosopher at a time. The footman process prevents all of the philosophers from being seated simultaneously, a well known deadlock-avoidance method.

A philosopher is specified by the following process (we use "$\oplus$" and "$\ominus$" to represent addition and subtraction respectively, modulo 5):

$$\begin{aligned}\mathrm{PHIL}_i = \epsilon \overset{[0,\infty)}{\rightsquigarrow} \mathrm{sit}_i \overset{[0,1]}{\rightsquigarrow} i.\mathrm{up\_fork}.i \overset{[0,3]}{\rightsquigarrow} i.\mathrm{up\_fork}.i \oplus 1 \overset{1}{\rightsquigarrow} \\ i.\mathrm{dn\_fork}.i \overset{0}{\rightsquigarrow} i.\mathrm{dn\_fork}.i \overset{0}{\rightsquigarrow} \mathrm{stand}_i \overset{1}{\rightsquigarrow} \mathrm{PHIL}_i\end{aligned}$$

Each philosopher may wait up to one second for his left fork and an additional three seconds for his right fork. After he stands, he must wait a minimum of one second before sitting down again. Notice that the fork processes have no bounds on the occurrence time of their events – their constraints are implicitly specified by their synchrony with the philosopher processes.

$$\begin{aligned}\mathrm{FORK}_i = (\epsilon \overset{[0,\infty)}{\rightsquigarrow} i \ominus 1.\mathrm{up\_fork}.i \overset{[0,\infty)}{\rightsquigarrow} i \ominus 1.\mathrm{dn\_fork}.i \overset{0}{\rightsquigarrow} \mathrm{FORK}_i) \Box \\ (\epsilon \overset{[0,\infty)}{\rightsquigarrow} i.\mathrm{up\_fork}.i \overset{[0,\infty)}{\rightsquigarrow} i.\mathrm{dn\_fork}.i \overset{0}{\rightsquigarrow} \mathrm{FORK}_i)\end{aligned}$$

We do not present the footman process here – for an untimed version one should consult [4].

The entire problem is now represented by the concurrent composition of the five forks, five philosophers and the footman: Dining_Phils = PHILS||FORKS||FOOT, where

$$\begin{aligned}\mathrm{PHILS} &= \mathrm{PHIL}_0||\mathrm{PHIL}_1||\mathrm{PHIL}_2||\mathrm{PHIL}_3||\mathrm{PHIL}_4 \\ \mathrm{FORKS} &= \mathrm{FORK}_0||\mathrm{FORK}_1||\mathrm{FORK}_2||\mathrm{FORK}_3||\mathrm{FORK}_4\end{aligned}$$

First, we claim without proof that the system's timing constraints are "intuitively" consistent. That is, if events are not arbitrarily delayed, deadlock cannot be an operational behavior.

We now present a scenario that ends in deadlock. Note that once a philosopher sits down, he does not get up without eating. If, however, he waits longer than 3 seconds for his right fork, he

stops. Furthermore, if he already had possession of his left fork, his stopping prevents his right neighbor from eating – the neighbor stops as well. Now recall that the the parallel operator does not *force* earliest synchronization of events. The result of this weak scheduling can be represented by the following trace:

$\langle\ (\text{sit}_0, 0), (0.\text{up\_fork}.0, 0), (\text{sit}_1, 0), (1.\text{up\_fork}.1, 0), (\text{sit}_2, 0), (2.\text{up\_fork}.2, 0), (\text{sit}_3, 0), (3.\text{up\_fork}.3, 0),$
$(3.\text{up\_fork}.4, 3), (3.\text{dn\_fork}.3, 1), (3.\text{dn\_fork}.4, 0), (\text{stand}_3, 0), (\text{sit}_4, 1), (4.\text{up\_fork}.4, 0)\rangle$

The deadlock that arises here is not the result of fundamentally inconsistent timing constraints, but instead of a parallel operator that is too weak. It puts no operational bounds on the waiting times for synchronizing events. In fact, they may arbitrarily wait up to their stated deadline, even if resources are available for them. A result of this problem is that the transition $i.\text{up\_fork}.i \overset{[0,u]}{\rightsquigarrow} i.\text{up\_fork}.i \oplus 1$ cannot be bounded for any u.

Taking the implicit approach, let us define a parallel operator that ensures event synchronization at the *earliest possible time.* The semantics of this operator then enforce the maximum parallelism paradigm. Furthermore, if we use these semantics in the method in the Dining_Phils problem, traces such as that illustrated above cannot occur, and the system is deadlock-free. It is indeed a difficult task to achieve the same set of system behaviors using an explicit scheduler. Its complexity is approximately that of the problem's state set – or 5^{11} transitions.

5 Future Work

We illustrated the effect of scheduling assumptions on specification. Our goal is to better understand the relationship between formal timing specification and assumptions about the underlying system such as hardware characteristics and scheduling algorithm. We are currently studying the relationship between explicit and implicit scheduling techniques for specification. We also are defining a hierarchy of implicit scheduling techniques, and investigating algebraic relationships between them.

References

[1] F. Jahanian and A. Mok, "Safety analysis of timing properties in real-time systems," *TSE*, vol. SE-12, pp. 890–904, September 1986.

[2] R. Koymans and R.K. Shyamasundar and W.P. de Roever and R. Gerth, and S. Arun-Kumar, "Compositional Semantics for Real-Time Distributed Computing," in *Logic of Programs Workshop '85, LNCS 193*, 1985.

[3] I. Lee and A. Zwarico, "An Algebra of Communicating Time Dependent Processes," Tech. Rep. MS-CIS-87-110, GRASP LAB 127, CIS Department, University of Pennsylvania, December 1987.

[4] C. Hoare, *Communicating Sequential Processes.* Prentice-Hall, 1985.

[5] A. Salwicki and T. Müldner, "On the Algorithmic Properties of Concurrent Programs," in *Proceedings of Logic of Programs, LNCS 125*, 1979.

The Integration of Deadline and Criticalness Requirements in Hard Real-Time Systems

Sara Biyabani *
John A. Stankovic
Krithi Ramamritham

Dept. of Computer and Information Science
University of Massachusetts

1 Introduction

Hard real-time systems are those systems in which the correctness of the system depends on both the *logical result* of the computation as well as the *time* at which such a result is produced. Hence, hard real-time systems are characterized by the constraints imposed upon the response time of the tasks in the system. It is crucial for the tasks in the system to meet their specified deadlines; the tasks may become worthless to the system if completed after the deadline, even if they are computationally correct. This strictness in meeting deadlines makes the proper scheduling of tasks a matter of great importance in the correctness and reliability of the system.

In the past, many *static* task scheduling algorithms have been developed, where the calculation of schedules has been done off-line using complete knowledge about the tasks' characteristics and arrival times. These algorithms have low run-time costs but they often fail to adequately adapt to changes in the environment. *Dynamic* scheduling algorithms, on the other hand, draw the task schedule dynamically at run-time and can therefore better respond to changes in the environment, but with increased run-time costs. However, the reliabilty and adaptiveness they provide outweighs the increased costs in most cases. In most previous work, tasks have been

*This paper was produced before this author joined Digital Equipment Corporation. The views expressed are exclusively those of the author and do not reflect the opinions or future product plans of Digital.

assigned 'priorities' which are implicit or explicit functions of their deadlines or *criticalness* or both. However, in actuality, these two requirements sometimes conflict with each other. That is, tasks with very short deadlines might not be very critical, and vice versa. We propose and analyze two distributed, dynamic scheduling algorithms that integrate criticalness and deadline such that, on a system-wide basis, not only do the most critical tasks meet their deadlines, but as many other tasks as possible also meet their deadlines.

2 The Algorithms

In this work it is assumed that each task has an associated criticalness and deadline and that we are dealing with a distributed system. In our approach we determine the schedulability of an incoming task as quickly as possible after its arrival so that a minimum amount of time elapses between the task's arrival and its dynamic, on-line guarantee. Both of the new algorithms first attempt to guarantee an incoming task according to its deadline, irrespective of its criticalness. If the task is guaranteed then the scheduling is successful.

However, if this first attempt at scheduling fails, then there is an attempt to guarantee the new task at the expense of previously guaranteed, but *less critical* tasks. If enough such lower criticalness tasks can be found then the new task is guaranteed at this site and the displaced tasks are transferred to alternative sites. If there are not enough less critical tasks, or the deadline of the new task is such that the displacement of any such tasks does not allow the new task to meet its deadline, then the *new task* is transferred to an alternative site. The above process is repeated at the next node until the task either meets its deadline or is discarded due to its deadline expiring. Hence, the notion of *conditional guarantee* lies at the heart of our algorithms. That is, a node is committed to the execution of a task it has *tentatively guaranteed* only so long as a more critical task does not arrive under such a pressing situation that in order to execute that more critical task, the previously guaranteed task has to be dislodged from the node.

The two new algorithms differ only in how they remove low criticalness tasks. In the first algorithm, lower criticalness tasks are removed one at

a time and in strict order from low to high criticalness. That is, all tasks at a particular criticalness are removed before any other tasks of a higher criticalness level. The second algorithm also only removes tasks of lower criticalness, but does not follow the strict order found in the first algorithm. Rather, it removes any task with lower criticalness, but chooses which specific task based on longest laxity. These two algorithms will be more fully described in the presentation.

2.1 Evaluation of the Proposed Algorithms

Since the problem of scheduling hard real-time tasks in a distributed environment is computationally intractable, we explored a simulation-based approach to analyze our algorithms. We considered how the overall system load and the balance of the system load amongst the nodes of the system affect the performance of the algorithms. We also studied how variations in the task-dependent parameters, the average laxity of the tasks and the numbers of tasks at different *criticalness levels*, affect the performance of the algorithms.

We compared the performance of our algorithms with that of four other baseline algorithms. Two of these baselines are deadline-based, one is criticalness-based, and yet another is a combined (deadline and criticalness) centralized algorithm that assumes perfect state knowledge. The performance of the two proposed algorithms is compared with that of the following baseline algorithms under identical conditions:

- DDLN, the deadline-based algorithm incorporating the notion of unconditional guarantee and a distributed load-sharing component.
- NC_CRIT, the criticalness-based algorithm without any node cooperation for load sharing.
- NC_DD, the non-cooperative deadline-based algorithm without any node cooperation for load sharing.
- C_CRDD, the centralized non-preemptive resume algorithm that keeps all the tasks in the system in a single queue ordered according to criticalness, and then deadline in case of ties.

3 Results

We find that combining deadline and criticalness does indeed result in improvements in the number of tasks executed in a real-time system. More specifically, from extensive simulations we show that:

- **Better Performance**
  Our proposed algorithms outperform all the other baseline algorithms (with the exception of the unrealistic bound of C_CRDD) under the range of system loads considered.

- **Adaptiveness Under Changing Circumstances**
  Under low load conditions, the deadline-based algorithms perform well, whereas under heavier loads, the criticalness-based algorithms tend to perform better. Our proposed algorithms outperform the baseline of DDLN in conditions of heavy load, and they outperform the baseline of NC_CRIT in conditions of low load.

- **Load Sharing vs. No Load Sharing**
  A major factor that affects the performance of the non-cooperative algorithms is the balance factor. The balance factor reflects the system condition where arrivals to the various nodes in the distributed system are not equal. When we compare the performance of non-cooperative algorithms with that of distributed algorithms that incorporate load sharing, we find that load sharing is a desirable feature of a scheduling algorithm in a distributed system environment. The need for load sharing is especially felt under low load situations. In these situations, tasks that are rejected from a busy site could be reallocated to the lightly loaded alternative nodes which exist in the system. Under high system load situations however, all nodes have an abundance of task arrivals and alternative sites for rejected tasks are not readily available.

- **A "Well-Balanced" and "Fair" Algorithm Desirable**
  From our studies on the centralized C_CRDD and the non-cooperative NC_CRIT algorithms, we conclude that there is a need to maximize not only the *worth* of the tasks guaranteed but also the number of

tasks guaranteed *at different criticalness levels.* A "fair" scheduling algorithm needs to avoid the potential for starvation of low criticalness tasks, by considering more than just the criticalness of tasks in determining the schedulability of tasks. The proposed algorithms achieve this fairness by disregarding the criticalness of an incoming task altogether in their first attempt at scheduling the task, when they consider only the deadline of the hard real-time tasks.

- **Conditional vs. Unconditional Guarantee**
  The simple deadline-based algorithm, DDLN, is a sufficiently good heuristic under non overload situations. However, its performance deteriorates very quickly in an overload situation. This can be attributed to its rigid notion of *unconditional guarantee.* In other words, this algorithm does not allow the "un-guaranteeing" of a task, whereas, our algorithms do.

  We can better understand the mechanics underlying our algorithms versus this often-studied deadline-based scheme if we separate the notions of *schedulability* and *serviceability* as applied to *guarantee.* The schedulability of a task depends on the conditions in the system when the task arrives at the node; the serviceability is a function of how the conditions in the system change during the life-time of the scheduled but not-yet-executed task. (If we allow preemption, then serviceability would depend on how conditions change between the time when the task is scheduled and when it *completed*, not started). In the case of an *uncondional guarantee*, a task's schedulability is the same as its serviceability. In cases of *conditional guarantee*, a task might be schedulable at first, but can become unserviceable as conditions in the system change. Hence, the conditional guarantee results in better adaptability of an algorithm to system conditions.

4 Conclusion

The results of our simulations show that our proposed algorithms perform better than the often-studied algorithms and, under a wide range of conditions, are very close to the ideal algorithm that assumes zero cost and

perfect knowledge about the state of all the nodes in the system. We show that combining deadline and criticalness does indeed result in improvements in the number and system value of tasks executed in a real-time system. Actual simulation results, in graph form, will be shown during the presentation.

5 Bibliography

- Jensen, E. D., Locke, C. D., and H. Tokuda, A Time Driven Sceduling Model for Real–Time Operating Systems, *Proc. Real–Time Systems Symposium*, pp. 112-122, December 1985.

- Ramamritham, K. and J. Stankovic, Dynamic Task Scheduling in Hard Real–Time Distributed Systems, *IEEE Software*, July 1984.

- Stankovic, J., K. Ramamritham, and S. Cheng, Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real–Time Systems, *IEEE Transactions on Computers*, Vol. C-34, No. 12, Dec. 1985.

Task Synchronization In Real-Time Operating Systems

Ragunathan Rajkumar
John P. Lehoczky
Departments of ECE and Statistics
Carnegie Mellon University

Abstract

Concurrent programming primitives for task synchronization and mutual exclusion include semaphores, monitors and the Ada rendezvous. However, a direct application of these primitives can lead to uncontrolled priority inversion, a situation in which a higher priority job is blocked by lower priority jobs for an indefinite period of time. In this paper, we investigate the class of *priority inheritance protocols* which solve the uncontrolled priority inversion problem. The *priority limit protocol* is an efficiently implementable priority inheritance protocol that minimizes the worst-case blocking time to the duration of execution of a single critical section. In addition, this protocol prevents the formation of deadlocks. If additional information is maintained, it is possible to enhance this protocol as well.

1. The Priority Inversion Problem

Scheduling jobs in hard real-time systems [1, 2, 3, 4, 6, 7] has been an important area of research. A recurring problem in this domain is the effect of blocking caused by the need for the synchronization of jobs that share logical or physical resources. Mok [5] showed that the problem of deciding whether it is possible to schedule a set of periodic processes is NP-hard when periodic processes use semaphores to enforce mutual exclusion. Mok [5] developed a procedure to generate feasible schedules with a kernelized monitor, which does not permit the preemption of jobs in critical sections. It is an effective technique for the case where the critical sections are short. Zhao, Ramamritham and Stankovic [9] investigated the use of heuristic algorithms to generate feasible schedules.

In this paper, we investigate the synchronization problem in the context of priority-driven preemptive scheduling, an approach used in many real-time systems. The significance of this approach is further emphasized by the fact that Ada, the language mandated by the US Department of Defense for its real-time systems, supports such a scheduling discipline. However, a direct application of synchronization mechanisms like the Ada rendezvous, semaphores or monitors can lead to uncontrolled priority inversion.

Priority inversion is said to occur when a higher priority job is forced to wait for the execution of a lower priority job. A common situation arises when two jobs attempt to access shared data. If the higher priority job gains access to the shared data first, the appropriate priority order is maintained. However, if the lower priority data gains access first and then the higher priority job requests access to the shared data, the higher priority job is blocked until the lower priority job completes its access to the data.

Example 1: Let J_1, J_2 and J_3 be jobs listed in descending order of priority. Assume that J_1 and J_3 share data guarded by a semaphore S. Suppose that at time t_1, job J_3 locks S and enters its critical section. During J_3's execution of its critical section, J_1 arrives at time t_2 and preempts J_3 and begins execution.

This work was sponsored in part by the Office of Naval Research under contract N00014-84-K-0734, in part by the Naval Ocean Systems Center under contract N66001-87-C-0155, and in part by the Federal Systems Division of IBM Corporation under University Agreement YA-278067.

At time t_3, J_1 attempts to use the shared data and gets blocked. We might expect that J_1, being the highest priority job, will be blocked no longer than the time for job J_3 to exit its critical section. However, the duration of blocking can, in fact, be unpredictable. This is because job J_3 can be preempted by the intermediate priority job J_2. The blocking of J_3, and hence that of J_1, will continue until J_2 and any other pending intermediate jobs are completed.

The blocking duration in Example 1 can be unacceptably long. This situation can be partially remedied if a job is not allowed to be preempted within a critical section. However, this solution is appropriate only for short critical sections. For instance, once a low priority job enters a long critical section, a higher priority job which does not access the shared data structure may be needlessly blocked. Analogous problems exist with monitors and the Ada rendezvous.

2. The Priority Inheritance Protocols

The use of *priority inheritance protocols* is one approach to rectify the priority inversion problem inherent in existing synchronization primitives. While our discussion of the protocols is based upon the use of binary semaphores, the techniques are equally applicable to monitors and the Ada rendezvous as well. We define a *job* as one instance of a task. Each job has the same priority as the task of which it is an instance of. We assume that jobs are scheduled according to the rate-monotonic scheduling algorithm, the optimal fixed-priority scheduling algorithm [4]. In the following discussion, job J_i has a higher priority than job J_{i+1}. The notation P_i represents the priority of job J_i.

The basic idea of priority inheritance protocols is that when a job J blocks higher priority jobs, it executes its critical section at the highest priority level of all of the blocked jobs. After exiting its critical section, job J returns to its original priority level. To illustrate this idea, we apply this protocol to Example 1. Suppose that job J_1 is blocked by J_3. The priority inheritance protocols stipulate that job J_3 execute its critical section at J_1's priority. As a result, job J_2 will be unable to preempt J_3 and will itself be blocked. When J_3 exits its critical section, it regains its original priority and will be immediately preempted by J_1. Thus, J_1 will be blocked only for the duration of J_3's critical section.

2.1. The Priority Limit Protocol

The priority limit protocol is a priority inheritance protocol with particularly good properties. It not only prevents deadlocks but also minimizes the worst-case blocking duration of each job to at most the duration of one critical section. The protocol is based on the idea that a lock on a semaphore is granted to a job if this critical section will execute at a priority greater than the priorities of all preempted critical sections, taking priority inheritance into consideration. If this condition is not met, the lock can be granted only if the priority of the new critical section is also its highest possible with priority inheritance *and* no deadlock or multiple blockings would result.

Definition: We define the *priority ceiling* (*priority floor*) of a semaphore as the priority of the highest (lowest) priority task that may lock this semaphore. The priority ceiling (priority floor) of a semaphore represents the highest (lowest) priority at which a critical section guarded by S will execute.

Example 2: Suppose J_1 accesses S_1 and then makes a nested access to S_2. Similarly, J_2 accesses S_2 and makes a nested access to S_1. Due to the possibility of mutual waiting, J_1 and J_2 can potentially deadlock. Note that the priority ceilings (floor) of S_1 and S_2 are equal to P_1 (P_2). At time t_1, let J_2 lock S_2. Before J_2 locks S_1, J_1 arrives and preempts J_2. However, when J_1 requests the lock on S_1, the run-time system finds that S_2 has been already locked by another job and the priority of J_1 is not greater than the priority ceiling of locked semaphore S_2. Also, the priority of J_1 is equal to the priority ceiling of S_1 but its priority floor

is not greater than the priority of J_2 which holds the lock on S_2. Since these two conditions fail, the run-time system denies J_1 the lock on S_1 and the deadlock is avoided. Instead, J_1 is blocked and J_2 inherits J_1's priority until it exits its critical section. Job J_2 then resumes its original priority, and J_1 can run to completion.

We now define the priority limit protocol.

1. Suppose that J has the highest priority among the jobs ready to run and is assigned the processor. Let S^* be the semaphore with the highest priority ceiling of all semaphores currently locked by jobs other than job J. Also, let the job holding the lock on S^* be J^*. Before job J enters its critical section, it must first obtain the lock on the semaphore S guarding the shared data structure. Job J will be blocked and the lock on S will be granted only if the priority of job J is higher than the priority ceiling of semaphore S^*, or if P = priority ceiling(S) and in addition, priority floor(S) > P^*. (The first condition is called the *priority ceiling condition* (*PCC*) and the second is called the *priority floor condition* (*PFC*)). In this case, job J will obtain the lock on semaphore S and enter its critical section. Otherwise, job J is said to be blocked by J^*. When a job J exits its critical section, the binary semaphore associated with the critical section will be unlocked and the highest priority job, if any, blocked by job J will be awakened.

2. A job J uses its assigned priority, unless it is in its critical section and blocks higher priority jobs. If job J blocks higher priority jobs, J *inherits* P_H, the highest priority of the jobs blocked by J. When J exits its critical section, it resumes its original priority. Priority inheritance is transitive. Finally, the operations of priority inheritance and of the resumption of original priority must be indivisible.

3. A job J, when it does not attempt to enter a critical section, can preempt another job J_L if its priority is higher than the priority, inherited or assigned, at which job J_L is executing.

The use of only *PCC* in this protocol yields the *priority ceiling protocol* [8]. If both *PCC* and *PFC* are *not* used and an unlocked semaphore can always be locked, we obtain the *basic priority inheritance protocol* [8]. We shall now illustrate the priority limit protocol using an example.

Example 3: Consider jobs J_1, J_2 and J_3. Job J_0 executes the steps { $\cdots$, P(S_0), $\cdots$, V(S_0), $\cdots$, P(S_1), $\cdots$, V(S_1), $\cdots$ }, job J_1 executes { $\cdots$, P(S_2), $\cdots$, V(S_2), $\cdots$ }, and job J_2 makes a nested semaphore access to S_1 by executing { $\cdots$, P(S_2), $\cdots$, P(S_1), $\cdots$, V(S_1), $\cdots$, V(S_2), $\cdots$ }. Hence, the priority ceilings (floors) of semaphores S_0, S_1 and S_2 are equal to P_0 (P_0), P_0 (P_2) and P_1 (P_2) respectively. Figure 2-1 depicts the sequence of events described below. A raised line indicates the job is executing, a flat line indicates the job has not arrived or has completed while shaded regions indicate critical section execution.

Suppose that

- At time t_0, job J_2 begins execution and later locks S_2.
- At time t_1, job J_1 arrives, preempts J_2 and begins execution.
- At time t_2, while attempting to access S_2 (already locked by J_2), job J_1 becomes blocked since both *PCC* and *PFC* fail. Job J_2 now resumes the execution of its critical section at its inherited priority of P_1.
- At time t_3, job J_2 successfully enters its inner nested critical section by locking S_1 since there is no semaphore S^* which is locked by jobs other than J_2. Job J_2 continues to execute at the priority of P_1, because this is the priority of the highest priority job currently blocked by J_2.
- At time t_4, job J_2 is still executing the inner critical section but the highest priority job J_0

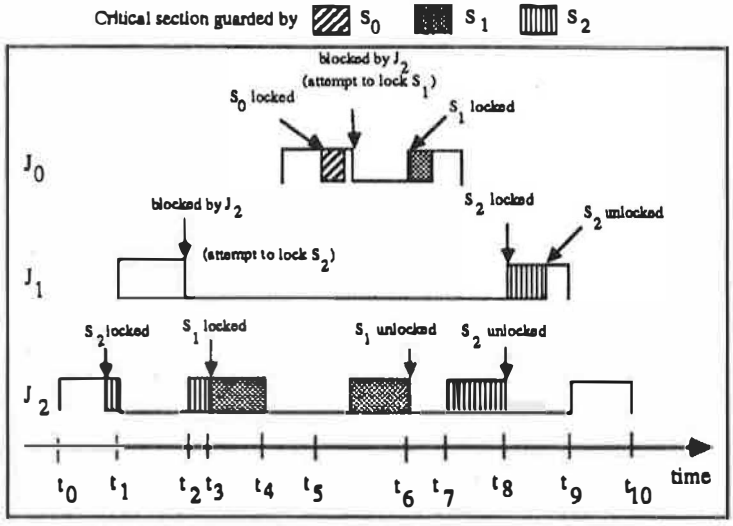


Figure 2-1: Sequence of Events described in Example 3.

arrives. Job J_0 preempts J_2, because its priority P_0 is higher than J_2's inherited priority P_1.

- At time t_5, job J_0 attempts to enter its critical section by locking S_0. Now, $S^* = S_1$ and $J^* = J_2$. J_0's priority P_0 is equal to the priority ceiling of S^* and hence *PCC* fails. However, P_0 = priority ceiling(S_0) *and* priority floor(S_0) > P_2. Thus, *PFC* succeeds and the lock is granted.
- At time t_6, job J_0 has released S_0 and then requests the lock on S_1. It can be seen that both *PCC* and *PFC* fail and hence the lock is denied and J_0 is blocked. At this point, job J_2 resumes its execution at the newly inherited priority level of P_0.
- At time t_7 job J_2 exits its inner critical section. Semaphore S_1 is now unlocked, job J_2 returns to the previously inherited priority of P_1, and job J_0 is awakened. At this point, J_0 preempts job J_2 and obtains the lock on S_0, because P_0 is higher than the inherited priority P_1 of job J_2. In addition, P_0 is higher than the priority ceiling of the only locked semaphore S_2, P_1. Job J_0 continues execution and later unlocks S_1.
- At time t_7, job J_0 completes its execution, and job J_2 resumes its execution of the outer critical section at its inherited priority of P_1.
- At time t_8, job J_2 exits its critical section, semaphore S_2 is unlocked, job J_2 returns to its own priority P_2 and job J_1 is awakened. At this point, job J_1 preempts job J_2 and J_1 is granted the lock on S_2. Later, J_1 unlocks S_2 and executes its non-critical section code.
- At time t_9, job J_1 completes its execution and finally job J_2 resumes its execution, until it also completes at time t_{10}.

It can be seen that both jobs J_0 and J_1 are blocked by the lower priority job J_2 for at most the duration of J_2's (outermost) critical section.

2.2. Enhanced Protocols

Though the priority limit protocol exhibits extremely desirable properties, it still remains a pessimistic protocol for it can permit blockings which can be avoided with additional information. Enhanced protocols, which explicitly consider the semaphores accessed by the preempted critical sections, can be devised. Such protocols can further reduce blocking but would require additional information to be maintained. However, a discussion of these protocols is beyond the scope of this paper.

3. Conclusions

Task scheduling in hard real-time systems has been an important area of research. In this paper, we have addressed the problem of task synchronization and mutual exclusion in the context of priority-driven preemptive scheduling. We showed that the direct application of primitives like semaphores, monitors and the Ada rendezvous can lead to uncontrolled priority inversion, a situation in which a higher priority job is blocked for a prolonged duration of time. We have investigated the class of *priority inheritance protocols* which solve the uncontrolled priority inversion problem. The *priority limit protocol* prevents the formation of deadlocks and minimizes the worst-case blocking duration to the execution time of a single critical section. Furthermore, the protocol can be implemented efficiently. The protocol can be used to determine schedulability and can be further enhanced by maintaining additional information.

References

[1] Lehoczky, J. P. and Sha, L.
Performance of Real-Time Bus Scheduling Algorithms.
ACM Performance Evaluation Review, Special Issue Vol. 14, No. 1 , May, 1986.

[2] Leinbaugh, D. W.
Guaranteed Response Time in a Hard Real-Time Environment.
IEEE Transactions on Software Engineering , Jan. 1980.

[3] Leung, J. Y. and Merrill M. L.
A Note on Preemptive Scheduling of Periodic, Real Time Tasks.
Information Processing Letters 11 (3):115 - 118, Nov. 1980.

[4] Liu, C. L. and Layland J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.
JACM 20 (1):46 - 61, 1973.

[5] Mok, A. K.
Fundamental Design Problems of Distributed Systems For The Hard Real Time Environment.
PhD thesis, M.I.T., 1983.

[6] Ramaritham K. and Stankovic J. A.
Dynamic Task Scheduling in Hard Real-Time Distributed Systems.
IEEE Software , July, 1984.

[7] Sha, L., Lehoczky, J. P. and Rajkumar, R.
Solutions for Some Practical Problems in Prioritized Preemptive Scheduling.
IEEE Real-Time Systems Symposium , 1986.

[8] Sha, L., Rajkumar, R. and Lehoczky, J. P.
Priority Inheritance Protocols: An Approach to Real-Time Synchronization.
Technical Report (CMU-CS-87-181), Department of Computer Science, CMU , 1987.

[9] Zhao, W., Ramamritham, K. and Stankovic, J.
Preemptive Scheduling Under Time and Resource Constraints.
IEEE Transactions on Computers , Aug. 1987.

PANEL

MYTHS IN REAL-TIME SYSTEMS --- FAIRNESS CONSIDERED HARMFUL

Chairman

John P. Lehoczky
Carnegie Mellon University

Panelists

Douglass Locke
IBM Federal Systems Division

Aloysius K. Mok
University of Texas at Austin

Lui R.Sha
Carnegie Mellon University

John A. Stankovic
University of Massachusetts

Pat Watson
IBM Federal Systems Division

NOTES:

NOTES:

SESSION II

Languages and OS

Chairman

Alan C. Shaw
University of Washington

ORE: Programming Real-Time Applications

David H. Jameson
IBM Research
P.O. Box 218
Yorktown Heights, NY 10598
dhj@ibm.com

Extended Abstract
ORE is a language and runtime environment designed for implementing high-speed real time systems. The testbed for ORE is a juggling robot. ORE and the robot are currently under construction in the System Structures group here at Yorktown.

ORE features include lightweight concurrency, low overhead, a type system, very efficient synchronization primitives, facilities for dealing with multiple processors and a restructurable scheduler. ORE is being implemented together with its own runtime environment designed to achieve the best possible performance for the system facilities provided by the language.

The Flavor of ORE

ORE is made up of a small number of basic components which are summarized here. These components and the justification behind their existence has been described elsewhere. [2]

Statement

A statement may be simple or compound. Examples of simple statements are assignments, **if then else**, **break**, **preempt**, **watch do** and **when do**. Sequences, strips and concurrences are compound statements.

Sequence

A sequence in ORE is a list of statements enclosed by angle brackets. A sequence executes as an infinite loop.

$$< S_1; S_2; \ ... \ S_n; >$$

A sequence may terminate itself explicitly by means of a **break** statement.

Strip

A strip is a sequence of statements that should be considered indivisible. Strips are enclosed in curly brackets. A strip is not an infinite loop.

$$\{ S_1; \ S_2; \ ... \ S_n; \}$$

Concurrence

A concurrence is a list of statements surrounded by square brackets. In this context each statement may be considered to be a process or task. A concurrence terminates when all the statements inside it have terminated. The statements are siblings of each other.

$$[\ S_1;\ S_2;\ \ldots\ S_n;\ ]$$

Sequences and concurrences can be nested inside each other to any level. A strip may be enclosed by a sequence or concurrence but cannot itself enclose a sequence or concurrence. A strip is therefore guaranteed to terminate within some given time. Strips, sequences and concurrences may contain their own local variables.

Preemption
Within a concurrence, a statement can preempt all its siblings. This allows a process to control other processes without any need for bookkeeping.

Watch and When
The **watch** and **when** statements have the following form where V is a list of variables, B is a boolean expression and S is a statement.

watch V **do** S
when B **do** S

Both of these statements suspend the calling process. In the case of a watch, the process is awakened when any of the variable being watched is assigned a value by some other process. The statement S is then executed as part of the wakeup. For a when, the process is awakened when any of the variables in the boolean expression is assigned a value. If the condition is not true, the process is suspended again. If the condition is true, the do part is executed.

In both cases, S may be a strip but it cannot be a sequence or concurrence.

Multiple Processors
ORE has adapted and extended the export/import mechanism found in languages like Modula-2 to support multiple processors. We allow compile time type checking for objects that exist in one processor and referenced by other processors.

Each processor has one main module, called a **machine** module. Machine modules may **import** code from **library** modules. The library module must have a corresponding **definition** module that can **export** the code. Library code is physically linked with all modules that import it. Library modules may import objects from other library modules.

A machine module wanting to call a process that resides on another machine must have a reference to that process at compile time to make the call possible. To differentiate between this and the import/export mechanism above, a see/expose mechanism is provided. A machine is said to **see** a process on another machine. For this to work, the other machine must **expose** such objects that it wishes the other machines to see. When a machine exposes a process, it exposes it to any machine that wishes to see it. It does not restrict the exposure to a specified machine. A machine module that exposes processes and variables has a corresponding **interface** module that declares the objects that are to be exposed.

Implementation Techniques
As ORE has evolved, many ideas have been thrown around in an effort to make the system as efficient as possible.

Task Switching
ORE uses non-preemptive scheduling. The compiler inserts codes at strategic points to suspend a process and transfer to the scheduler. At these context switch points, it is known that no state information will be found in registers (with the exception of the stack pointer) and therefore process switching is very cheap. This mechanism is derived from work done on the Pluribus multiprocessor [3], Tomal [4], and Owl [1,5].

Stack Allocation
Processes in ORE are very lightweight, numerous and often live for very brief periods. For each process that is created, a stack must be allocated that will contain local variables, space for watch structures and the activation record. One way to do this is to have a dynamic stack allocation mechanism. With this mechanism, as part of the creation of a new process, a request is made for a suitably sized stack. A stack manager allocates and deallocates stacks as required. The stack manager must be able to provide stacks of any requested size and deal with problems such as fragmentation.

However, it is possible to take advantage of the characteristics of ORE processes so that a much cheaper static mechanism can be found. For ORE processes, we can determine how much memory will be needed at compile time and generate code to allocate memory once at the outermost level.

Consider the following segment of code whoses processes have been named for reference. The numbers indicate the amount of space needed by each process for its own local use.

```
<                   -- A, 8
  [                 -- B, 4
    < >             -- C, 2
    < >             -- D, 6
    < >             -- E, 10
  ]
>
```

A is a sequence containing a concurrence B that contains three sequences C, D and E. It is known at compile time how much stack space will be needed for each of A, B, C, D and E.

We allocate a 30 byte stack from a stack manager when A is created. When B is created, we can allocate its stack simply by adding 8 to the current stack pointer. C, D and E are created at the same time. Their stacks will start at offset 12, 14 and 20 respectively. Note that if B had been a sequence then we would only have neeeded to allocate 22 bytes at point A because C, D and E could share the same space since only one of them can exist at any time.

This scheme will not work if recursion or dynamic spawning of processes is allowed. However, recursion is of limited use in the real-time robotics domain. If it is needed, a dynamic stack allocation scheme must be used and the programmer must be prepared to pay the extra cost.

References

[1] M. D. Donner
Control of Walking: local control and real-time systems.
PhD thesis, Carnegie-Mellon University, 1984

[2] M. D. Donner, D. H. Jameson
Language and operating system features for real-time programming.
Computing Systems, Vol 1, No. 1, 1988

[3] S. M. Ornstein et al
Pluribus - A reliable multiprocessor
AFIPS 1975 Conference Proceedings

[4] J. L. Henessy
A Real-Time Language for Small Processors: Design, Definition, and Implementation
PhD thesis, SUNY Stony Brook, August 1977

[5] M. D. Donner
The Design of OWL: a language for walking
Proceedings of the Sigplan 1983 Symposium on Programming Language issues, ACM June 1983

Refinement and Enhancement: Primitives for Monotonic Computations

Kwei–Jay Lin and Swaminathan Natarajan

Department of Computer Science
University of Illinois at Urbana–Champaign

1. Introduction

Most real–time systems have a static configuration with limited resources. To provide graceful degradation during dynamic or faulty situations, it is desirable for programs to continue functioning despite some resources not being available. In real–time systems, time for program execution may be viewed as a resource whose availability varies from execution to execution. Like with other resources, we may want to maintain the operation of a system when the execution time available is less than that is normally required.

Flexibility in execution time requirements can be obtained by allowing a computation to return results of different precisions. If enough time is available, the computation will produce precise results. If not, shorter versions of the computation are executed to produce approximate, *imprecise* results. For such a scheme to work, computations must be *monotonic*, i.e. the results produced by the computation should be more precise as more time and resources are consumed. In this paper, we present two new language primitives, *refinement* and *enhancement*, which may be used to implement monotonic computations. The primitives are part of the language FLEX being developed in the Concord project [Lin87a, Lin87b].

2. A simple example

To motivate our research, we first discuss how a simple real–time program can be implemented. The program is to receive the vision signal from a robot and produces movement instructions to steer the robot periodically. The system must recognize the obstacles on the field and find a route to get around them. Part of the program is as follows:

```
loop every 5 sec
        {
        signal := receive();          /* receive vision signal from robot */
        layout := pattern(signal);    /* interpret the objects */
        route := plan(layout);        /* find the next move */
        send(route);                  /* send the instruction to robot */
        }
until layout = destination;
```

This work was supported in part by a contract from the ONR (N00014–87–K–0827).

To always meet the deadline, most other systems would adopt a pessimistic approach: making sure that even in the worst case, the execution time for each iteration is less than 5 seconds. The approach has two drawbacks. First, the system implemented is usually very under–utilized, since the worst–case scenario is used as the system specification. The system is also vulnerable to hardware failures.

Our proposed solution to the problem is to implement the procedures so that they can produce results with different degrees of precision, depending on the length of time executed. For example, *pattern* can do a precise matching or a rough matching, and *plan* can generate a complete plan or just a temporary action. The procedures are implemented to produce an approximate result first, and then monotonically improve the results using more time and resources. Thus if there is no time to produce the most precise result, some basic or inferior result is still available. The system can be designed to handle only normal loads so that the resources are efficiently utilized most of the time.

To provide a result with imprecision, FLEX allows a value in computations to be an *interval* rather than an exact number. An interval is used as a number with imprecision. FLEX provides the *refinement* operation which always reduces the imprecision (or *range*) of an interval by intersecting the old value with a new value. Refinement is preferred to assignment since it guarantees the monotonic improvement of the precision in a value. Thus if more time or resource is available, more refinements can be performed, and more precise results can be produced. Another way to improve the precision is to dynamically *enhance* a computation with some extra operations when more time is allowed for execution.

3. The concept of precision in FLEX

3.1. Values

FLEX defines *primitive* values which are similar to those we are familiar with, such as integers, real numbers, characters and constants. In addition, FLEX also manipulates *derived* values which represents alternative possibilities of primitive values. For example, the derived value [2|3|4] represents a value which is either 2 or 3 or 4. A derived value may also be expressed in terms of a range. The derived value [1..7] represents a value which lies between 1 and 7. *A derived value is not a set of values.* It represents alternative possibilities for a single primitive value that is not yet identified.

A type in FLEX consists of a set of primitive values and all the derived values which can be constructed from them. This includes two special derived values, $\top$ and $\bot$. $\top$ represents "any primitive value in the type" and $\bot$ represents "none of the primitive values in the type". In fact, each type has a $\top$ and $\bot$, which should be written as $\top_{int}$, $\top_{bool}$ etc. We will omit the subscripts and leave them to be interpreted from the context. Syntactically, a type is defined by specifying its primitive values, as below:

type bool: {*true*, *false*};

The primitive and derived values in a type constitute a lattice of values [Gierz80,Scott76]. The lattice is based on the relation $\mathcal{R}$ (for $\mathcal{R}efined$) defined next. The notation "$a \subset m$" in the definition means that a is a member of the set of possibilities represented by m.

Definition: The relation $\mathcal{R}$ maps a value m to another value n if and only if every possibility included in m is also included in n, *i.e.* $m \mathcal{R} n$ iff $a \subset m$ implies $a \subset n$.

For example, [1|2] $\mathcal{R}$ [1|2|3], and [1] $\mathcal{R}$ [1|2]. We can show the values in any type form a lattice by observing that $\mathcal{R}$ is reflexive, symmetric and transitive. Intuitively, $\mathcal{R}$ defines the "more precise than" relationship. If $m \mathcal{R} n$, m is a *more precise* value than n, in that there is less uncertainty about the value represented.

3.2. Functions

Functions in FLEX accept a list of input parameters and produce a list of outputs. Functions are entirely stateless. Given a function *square_root* and an input 4, the function produces the output [–2|2]. This example illustrates that the functions in FLEX can map a single input into several alternatives.

Similar to the lattice of values, each function actually forms a lattice. Given the same input x, suppose a function f produces an output y, and another function g produces y' as output. g is *more precise* than f if and only if the outputs of g are always more precise than those of f for all inputs. We also define functions $f_{\top}$ and $f_{\bot}$, which are functions always producing $\top$ and $\bot$ respectively as output regardless of what inputs are given. Functions which work on the same input and produce outputs of different precisions form a lattice according to the precision of the results.

4. Monotonic primitives

Programs in FLEX consist of a set of *objects*. Each object consists of *statics* and *procedures*. The statics carry state information, and the procedures cause state transformations. Procedures and functions can also have variables, or *locals*, defined internally. Locals and statics are generically referred to as *items*.

4.1. Refinement

Refinement is an alternative way to assignment in modifying the value associated with an item, by making the value more precise rather than assigning an entirely new value to the item. Refining an item x with another item y reduces the possible values of x to those which are also possible values of y. For example, if x is [1|2|3] and y is [2|3|4], refining x with y causes x to get the value [2|3]. It is analogous to set intersection. Refinement of x with y is written "$x \leftarrow y$".

Refinement is a very useful operation, because it performs computations without using "variables" which change values. Since refinement does not introduce new possibilities, assertions previously made about all possible values of a local continue to be true after a refinement. FLEX views computations as a process of reducing the imprecision in the results until a precise enough value is obtained. Accordingly, refinements are preferable to assignments in monotonic computations. Assignment is used for updating temporary locals, such as the loop index in an iteration, as well as statics. Updating a static is viewed as a state transition which is necessary in many systems.

4.2. Enhancement

To make a function more precise, many strategies are possible. Algorithm, time and resources are the key parameters involved. A real–time function usually has no control over the time and resources available during its execution. It can only employ different algorithms to achieve different precisions. Our objective is thus to modify the algorithm on the fly so that a function can dynamically adjust to its execution environment. We call such modification *enhancement*.

One way to implement enhancement is to invoke *sieves*. A sieve is a function whose output–list is identical to the input–list and each output value is a refinement of the corresponding input value. Thus the only purpose of a sieve is to increase the precision of the inputs. Given a function with several sieves, we can enhance the function by including the invocation to one or more sieves at run time.

Sieve functions are actually not uncommon. Many numerical methods use iteration to obtain more precise results. Each iteration can be regarded as a sieve. We are still investigating other forms of enhancement.

4.3. Implementation issues

Imprecise values may be either a discrete set of possibilities or a continuous range. Discrete possibilities are manipulated by performing the computation on each of the possibilities. When there are many possibilities, this can result in substantial overhead. But exploring each of multiple discrete possibilities is to increase the precision in our knowledge; thus the effort is not really wasted. If vector processing hardware is available, we can manipulate all the possibilities simultaneously.

Ranges are useful whenever a computation is performed on real numbers. Performing computations on ranges involves *interval arithmetic*, which has been the subject of much study[Moore66, Spr75, Ratschek84]. While for some operations such as addition it is sufficient to perform the computation on the end points of the range, for other operations more complex techniques are necessary. For applications involving extensive manipulations on intervals, it is possible to utilize special hardware to perform interval arithmetic.

In [Lin87b] we presented a scheme to support enhancement by using a "supervisor" for each imprecise function. If no termination request is received by the supervisor, a function would execute all codes, including all sieves, as defined to acquire the best precision. But if the supervisor is notified of an immediate termination, the function then skips certain sieves to produce an imprecise result.

5. Conclusion

Two unique primitives in FLEX were presented. One is the ability to refine imprecise values. While it is possible to create appropriate data structures and operations to simulate this in other languages, incorporating imprecision in the language model makes the semantics of the refinement clearer. This in turn makes it easier to provide special support at the architecture and system level, and simplifies program validation. Another unique feature of FLEX is enhancement. Certain segments of codes can be selectively executed if time and resources required are available.

The emphasis in the design of FLEX is flexibility. The burden of meeting all deadlines is shared between the programmer and the run–time system. The programmer designs an algorithm which works monotonically towards a precise result. By doing this, the flexibility needed for real–time executions is obtained.

References

[Gierz80]
Gierz, G. *et al*, *A Compendium of Continuous Lattices,* Springer–Verlag, Berlin, 1980.

[Lin87a]
Lin, K. J., S. Natarajan, and J. W. S. Liu, "Concord: A system of imprecise computations," *Proc. COMPSAC87,* pp.75–81, Tokyo, Japan, October 1987.

[Lin87b]
Lin, K. J., S. Natarajan, and J. W. S. Liu, "Imprecise results: Utilizing partial computations in real–time systems," *Proc. Eighth Real–Time Systems Symposium,* San Jose, CA, pp. 210–217, Dec. 1987.

[Moore66]
Moore, R.E., *Interval Analysis,* Prentice–Hall Inc., Englewood Cliffs, N.J., 1966.

[Ratschek84]
Ratschek, H., and J.Rokne, *Computer Methods for the Range of Functions,* Ellis Harwood Ltd., Chichester, UK, 1984.

[Scott76]
Scott, D., "Data types as lattices", *SIAM J. Comput.,* vol.5, No.3, pp.522–587, 1976.

[Spr75]
Interval Mathematics, Lecture Notes in Computer Science, vol.29, Springer–Verlag, Berlin, 1975.

CHAOS and CHAOS-ART - Extensions to an Object-Based Kernel

Ahmed Gheith
Prabha Gopinath
Karsten Schwan
Peter Wiley

The Ohio State University
1988

1 Introduction

CHAOS is an object-based operating system [1, 5, 3, 4] tailored specifically towards the real-time domain. Using CHAOS, application software can be organized as a collection of interacting objects. CHAOS allows objects of varying weights and visibility to be constructed. In addition, it supports a comprehensive set of interaction primitives. Previous work in CHAOS has been reported elsewhere [7]. We describe further research in CHAOS and its suitability for building complex real-time applications. We use a complex robot tracking application to illustrate the strengths and weaknesses of CHAOS and to motivate CHAOS-ART, an extension to CHAOS that supports nested atomic actions as the basic mechanism for synchronization and recovery.

2 A Robot Tracking Application

The robot tracking program controls a conveyor, vision system, and robot arm.It executes on a testbed consisting of an eight-node bus-based multiprocessor and a Sun workstation.

The vision system is used to detect and determine an initial reference position for a block on the conveyor, whose speed is determined by the conveyor interface. Given these two inputs, the software generates the control signals needed by the robot arm to track, grasp, and remove the block from the conveyor. Additional blocks will be removed, and stacked on the original part. Limitations in the vision system and robot combination make it impossible to obtain continuous position feedback of the robot's end-effector in relation to the part being tracked. Also, the robot arm's inherently high latency of mechanical response requires that the part being tracked be picked up within at most one conveyor revolution. After this period of time, the positions computed for the end-effector differ significantly from the actual position of the part. Failure to acquire the part within this time frame implies a tracking failure; the arm then resets itself and the process repeats. Thus the tracking application is bounded by a hard deadline - the time required by the robot to grasp the object, which in turn corresponds to the time required by the conveyor to complete one revolution.

The application's thirteen CHAOS objects have been abstracted into four groups on the basis of their functionality. These are (1) low-level objects, (2) mid-level objects, (3) high-level objects, and (4) database objects. The complete object structure of the tracking application is shown in Figure 1.

High-level objects implement user interfaces and operations to interpret the data produced by the Mid-Level Objects. The objects in this group are:

- DriverObject - this object accepts user inputs, sets in motion the system initialization sequences and conveyor, and invokes the HighLevelPlanObject to initiate the tracking.
- HighLevelPlanObject - initiates the tracking algorithm by invoking the SensorObject to determine the number of parts on the conveyor. Then PartObject is invoked to produce the position with respect to time for each part. The translational and rotational velocity vectors of the conveyor are accessed from the DataObject. Finally, TrackObject is invoked to determine the projected position and time for grasping the part.

Mid-level objects arithmetically manipulate the raw data obtained from the low-level objects and execute the decisions made by the high-level objects. These are:

- LowLevelMotionPlanObject - receives the desired position and original joint angles from MoveObject. LowLevelMotionPlanObject then determines the necessary joint velocities needed to reach the final position and invokes InverseJacobianObject to produce the corresponding joint rates.

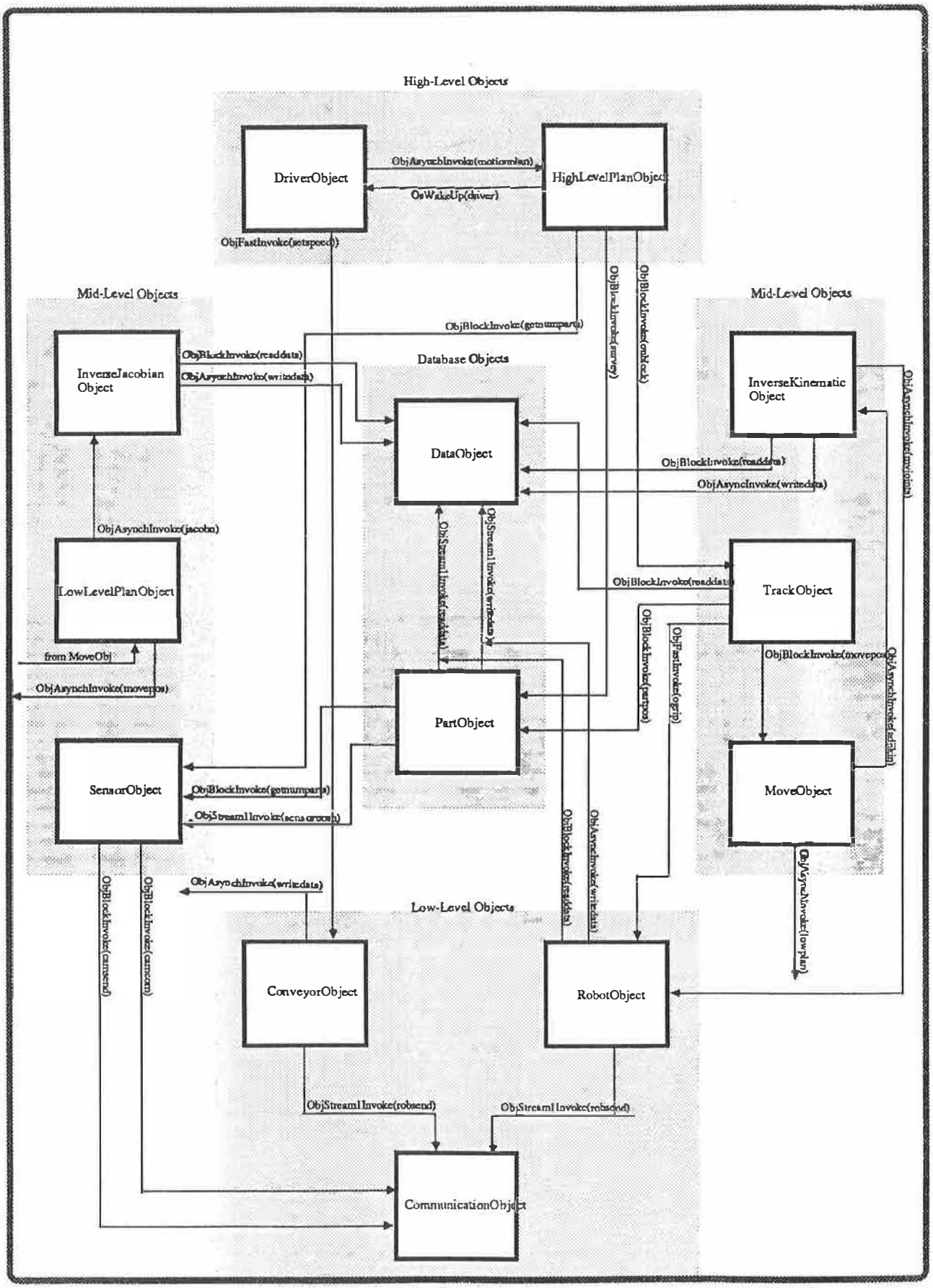


Figure 1: A Robotics Tracking Application

- TrackObject - receives the position of the part at a given time and the translational and rotational velocity vectors of the conveyor. It then computes the projected end-effector position and the time to grasp the desired part. Additionally, it periodically invokes PartObject to (1) update the number and position of the parts on the conveyor and (2) determine the part's projected position at a given time from which it bases its decision on the end-effector. Lastly, MoveObject is invoked to initiate the sequence of steps needed to move the arm to the projected position.
- MoveObject - receives the world coordinates of the point to which the robot arm end-effector is to be moved. MoveObject invokes the InverseKinematicObject to determine the corresponding joint angles.
- SensorObject - contains control operations for the low-level vision system interface contained in the CommunicationObject. SensorObject determines the number of parts and their position on the conveyor.
- InverseKinematicObject - like InverseJacobianObject, is a computation intensive object. Given the desired position vector of the end effector in the world coordinate system, the InverseKinematicObject will determine the necessary joints angles to achieve this position. Then RobotObject is invoked with the final joint angles to move the arm and end-effector.
- InverseJacobianObject - given the joint velocities by the LowLevelPlanObject necessary to reach a position in the reference coordinate system, InverseJacobianObject determines the corresponding joint rates by calculating the Jacobian and its inverse for the current joint angles using the inverse Jacobian matrix.

Database objects store attributes of the blocks on the conveyor belt and other system attributes:

- PartObject - invokes SensorObject to sense the attributes of the parts on the conveyor that are within the field of view of the vision system; it also provides the projected position of the part to TrackObject. SensorObject returns the position of the parts with respect to the vision systems coordinate frame; PartObject converts this position into the robot arms coordinate system. Generated data (position, last update time, etc.) are stored in the DataObject database.
- DataObject - A database object to store information written and read by numerous objects, especially information about the parts viewed on the conveyor, and matrices used both by InverseKinematicObject and InverseJacobianObject.

Low-level objects interface directly with the hardware present [robot arm, vision system, and conveyor] to generate raw data:

- RobotObject - receives final joint angle data from the InverseKinematicObject, then calculates the actual low-level commands to send out to the robot to control its movement.
- ConveyorObject - accepts commands to start, stop, or regulate the speed of the conveyor. Invokes DataObject to update this information in the database.
- CommunicationObject - contains interface operations directly to the resident hardware. Controls and monitors the serial lines between the multiprocessor and the vision system and robot arm.

3 Ramifications on the Design of CHAOS

Based on our experiences with the tracking program, we designed and incorporated several changes in CHAOS.

Queueing primitives: The first version of CHAOS supported a variety of queueing paradigms for invocations. These included a priority queue, a deadline queue, and a FIFO queue.

The FIFO queueing mechanism is easily simulated with the deadline queueing mechanism simply by setting the invocation deadline to Infinity. Since CHAOS orders deadline requests by earliest deadline first, such requests are inevitably serviced after all other requests in the system, and since CHAOS maintains a FIFO ordering among invocations tagged with identical deadlines, the FIFO ordering is maintained among all requests with deadline set to Infinity.

As regards priority queues, their primary usefulness lies in the fact that they are intuitively easier to use than deadlines. However, in practice there are several difficulties associated with using priorities to determine the sequence in which invocations should proceed. Since the value of the priority assigned to an invocation is an extremely subjective choice, determining a precise, repeatable sequence of invocations became a difficult task. Again, most applications are bounded by a hard overall deadline. For example, in the tracking application, all computations associated with determining the final position and velocity of the robot must be completed within a given deadline, which in this example is the time taken by the conveyor to move the part to the "pick-up" point. Generating a schedule of priority invocations in order to meet a such deadline, is a difficult task. As a result we decided to limit the scope of our priority queueing mechanism. Instead of ten levels of priority, we elected to provide a notion of *criticality* that could then be used to characterize each invocation in the system. An invocation could now be tagged as being "critical" or "non-preemptible", indicating that it could not be *preempted* by another invocation, whereas a "non-critical" or "preemptible" invocation may be preempted to satisfy the deadlines of some other "critical" invocation.

It is our contention that deadline-based schedulers and priority-based schedulers cannot co-exist. The problem associated with making multiple levels of priority available to a queued invocation is that it becomes difficult to guarantee deadlines for invocations. Consider a scenario where an invocation with a lower priority has been accepted for service. In CHAOS such an acceptance constitutes an implicit guarantee that the requested deadline of the request will be satisfied. Suppose now that while the low-priority request is executing, an invocation with a higher priority arrives. If the priority-based object scheduler is currently active, then the request will be accepted and allowed to preempt the executing operation, possibly violating the guarantee offered earlier. If however the deadline-based scheduler is currently active, then service for the new request will be deffered till after the current request has completed. We believe that our solution of assigning a simple critical/non-critical attribute to each invocation merges the intuitive appeal of priorities with the convenience and predictability of a deadline scheduler.

Based on the conclusions drawn above, CHAOS has been redesigned to allow only deadline-based queued invocations. All invoked operations execute at a fixed and identical priority level. Certain invocations can be marked as being more critical than others by tagging them with the "non-preemptible" flag.

Invocation synthesis: CHAOS provides five different non-blocking invocation primitives. These are the ObjFastInvoke(), ObjInvoke(), ObjAsynchInvoke(), ObjStream1Invoke(), and the ObjStream2Invoke(). Of these, the ObjFastInvoke(), ObjInvoke(), ObjStream1Invoke(), and ObjAsynchInvoke() are extensively used in the tracking application. However, it was soon evident, that a system that provided only a fixed set of invocation primitives would be too restrictive. We thus extended our earlier work of decomposed invocation primitives [7] and implemented a mechanism whereby an application programmer could specify and synthesize an invocation primitive with the functionality and performance characteristics that

he desired. For example, in the tracking application, it was found desirable to have an invocation primitive that had the functionality of the ObjFastInvoke() but also maintained status of the invocation. To aid the application programmer in synthesizing new invocation primitives, standard building blocks have been implemented. These building blocks completely characterize any invocation primitive. Compositions of these building blocks yield invocation primitives with differing functionality and performance.

Guaranteeing performance: In any real-time application, raw speed of execution can be achieved without compromising predictability and accountability. Therefore, in the context of objects and invocations, acceptance of an invocation by the target object constitutes an implicit assurance that the deadline of the invocation will be satisfied. CHAOS provides mechanisms to ensure that this criterion is satisfied. CHAOS also relieves the application programmer of the task of computing the intermediate deadlines that must be satisfied for the goal deadline(s) to be met. Specifically CHAOS stores a complete representation of the invocation graph of an application. Given the start time of the application and the goal deadline(s), intermediate deadlines for all invocations in the graph are computed. If for some reason, the goal deadline(s) change, the invocation graph is recomputed to yield new values for the intermediate deadlines. Another facility incorporated in CHAOS is that of actually adapting the invocation primitives themselves. The scope of dynamic deadline graph recomputation is limited by the overall load on a processor node. When it is no longer possible to generate a schedule of invocations within the constraints of the specified goal deadline, CHAOS allows invocation primitives and their invoked operations to be modified. Thus in the case of the tracking application, it becomes feasible to temporarily substitute a compute-intense version of the InverseKinematicObject with one that is faster, but performs a simple extrapolation of the robot's current position.

Windows: The Windows mechanism in CHAOS supports shared-memory interactions among objects [2], and in addition offers the advantages of controlled access to shared space. Such interactions were found to be particularly useful between the SensorObject and the PartObject, and between the DataObject and the PartObject. Windows allow an object to define portions of its internal state as being *externally visible*. Other objects can then open either a WriteWindow or a ReadWindow on such "visible" state. Any writes into the WriteWindow modify the windowed areas of the target object's state. Reads from a ReadWindow, will display the contents of the windowed areas of the target object.

4 The CHAOS Programming Environment

The CHAOS environment integrates a high-level language interface to CHAOS, an augmented entity realtionship representation framework that stores objects, operations, interactions, and real-time attributes of CHAOS applications as entities and relationships and a monitoring and remote invocation facility. This uniform framework serves as a complete repository for all information about the application, and may be accessed by the Adaptation Controller, the Monitoring Mechanism, and the Adaptation Enactment mechanism. Owing to the support that the CHAOS environment provides for specifying and enacting adaptations on the basis of high-level user specifications, it may more properly be termed an *adaptation environment*.

5 CHAOS-ART: Atomic Transactions in CHAOS

CHAOS-ART is an extension to CHAOS that supports nested atomic actions as the basic mechanism for synchronization and recovery. Our experiences with the ASV, the tracking application, and CHAOS, has shown that atomicity fits well with the concurrency control and recovery requirements of real-time systems. However, implementations for database applications or distributed computing&cite{Weihl85}, contradict a real-time system's requirements of concurrency, responsiveness, urgency, and high performance. Specifically, traditional definitions of concurrency atomicity assume the scheduling of concurrent activities to enforce a serialized order (e.g. Two-phase locking). This is undesirable in real-time systems because it reduces potential concurrency. As a result, the system will not always be able to respond to asynchronous events if the responding activity has to compete with other ongoing activities. This reduces the reponsiveness of the system and may disturb execution scheduling required by the urgency hierarchy.

Similarly, regarding recovery of real-time actions, such recovery cannot always be achieved by rolling back to some previous "consistent" state because: (1) Time can not be rolled back. The time spent in doing the (partial) action can't be "unspent". (2) If an action affects the external environment in any way, it might be impossible to undo that effect (some physical processes are irreversable, e.g. launching a rocket, moving a car in a one way street, etc.). (3) If the action responds to external events, undoing it implies either loosing those events or regenerating them. Loosing the events means that the action is not completely undone while regenerating them means that they were delayed as a side effect of the failed action. In both cases, the action is not really undone.

To compensate for the potential degradation in concurrency of strongly serialized atomic actions, CHAOS-ART relaxes the strict two-phase locking scheme allowing the programmer to release locks before the action terminates, provided that it would not result in an inconsistent state. Responsiveness to asynchronous events is improved by using *revocable locks*. Revoking a lock from an action can either abort the action (atomic) or delay it until the lock is returned (non-atomic). The system does not guarantee the consistency of shared resources with non-atomic locks. By revoking a lock, an "urgent" action can preempt less urgent ones competing for the same lock. Thus, enforcing the required hierarchy of urgency levels.

Since backward recovery of aborted actions is not always possible, CHAOS-ART supports both backward and forward recovery (Compensatable Objects [6]). Backward recovery is provided automatically by the system; the recoverable state is automatically restored when an action aborts. Forward recovery is provided by the programmer in the form of a *compensation operation* to be executed in case of abortion.

Pre-scheduling provides a "practical" compromise between the high overhead associated with atomic objects and the high performance often required by real-time systems. It also helps in absorbing transient overloads by anticipating future activities and preparing for them ahead of time. Pre-scheduling uses the slack time in most applications executions to distribute the additional overhead imposed by atomic actions. Activities that can be pre-scheduled include: (1) Lock acquisition: the locks that are needed by an invocation may be acquired before the invocation actually starts. Lock definition has been extended to allow locks to be *pre-acquired*. A pre-aquired lock is kept by an activity as long as no other activities are waiting on it. If a running activity tries to acquire a pre-acquired lock, the lock is granted immediately. If another activity tries to pre-acquire it, the scheduler decides which activity can keep the lock. The

decision depends on the scheduling policy being used. A reasonable policy is to grant the lock to the activity with the closest deadline. (2) Copying the recoverable state: a delayed invocation tries to keep a recent copy of the recoverable state. The object's scheduler maintains a list of all invocations with copies of the recoverable state. If it changes, the scheduler invalidates the copies and starts updating them. (3) Pre-scheduling invocations: invocations within the body of a delayed object operation are recursively pre-scheduled. Information about these invocations can be either provided by the programmer or automatically generated by the language processor.

Objects in CHAOS-ART can be either atomic or non-atomic. For *NonAtomic* objects, the system does not provide any support for the consistency of the object's state. It is the responsibility of the object's programmer to make sure that it can withstand arbitrary failures and to synchronize parallel invocations. Non-atomic invocations always succeed.

Atomic objects provide some degree of serialization and failure recovery. In general, atomic objects can be made fully serializable at the expense of potential concurrency by using strict two-phase locking. Each operation specifies a set of locks that are to be automatically acquired before starting the invocation. These locks are not released at the end of the invocation. Instead, the system keeps a list of these locks and automatically releases them at the termination (success or failure) of the action doing the invocation. Full serialization can be achieved by using an exclusive lock and requiring each operation to acquire that lock at the beginning. This scheme, however, does not allow any concurrency at all. Knowing the semantics of the object, the programmer can define a combination of locking patterns that will allow more potential concurrency and still maintain the consistency of the object.

Two recovery schemes are provided for atomic objects: forward and backward recovery. **Forward** recovery is defined by the programmer by defining a recovery procedure for each recoverable action. The recovery procedure should semantically undo the effects of the action. **Backward** recovery is similar to the conventional recovery provided in data base systems and is achieved by splitting the state of each atomic object in two parts: **recoverable** and **non-recoverable** state. A single copy of the non-recoverable state is shared by all invocations which means that the changes made to it can't be automatically undone (they can be undone, however, by the forward recovery procedure). The recoverable state, on the other hand, is not shared by multiple invocations of the same object. Instead, each invocation manipulates a local copy of the recoverable state that is discarded if the invocation aborts. If the invocation commits, the local copy becomes the current object state and the old state is discarded. Full backward recoverability can be achieved by making the whole state recoverable.

Invocations: An atomic invocation can have a deadline, a delay, a start condition and a stop condition. Atomic invocations are implemented in four parts: The **Prologue**, **Body**, **AntiBody** and the **Epilogue**. The **Prologue** can be supplied by either the programmer or the system. It specifies the activities to be performed before entering the body. There are two kinds of Prologue activities: lock acquisition and specification of pre-scheduled invocations. The Prologue is separated from the Body for efficiency reasons; while the body is waiting to start, the Prologue can start acquiring locks and creating templates for future invocations.

The **Body** is provided by the programmer. It is activated by the system after both the delay specified in the invocation expires and the start condition is enabled. The body can be interrupted if the stop condition is enabled before it commits (failure).

The **Anti-Body** is an optional forward recovery procedure that is provided by the programmer. It is activated if the body fails. The anti-body has the same view of the object's state as the Body.

The **Epilogue** is provided by the system. If the action fails (aborts), it is activated after the anti-body is executed. On the other hand, if the action succeeds (commits), it is activated after the top level action succeeds (commits). The Epilogue has the task of cleaning up after the action. For an aborted action, it releases all locks and aborts all siblings. For a committed action, it updates the recoverable state, commits its siblings, and releases all locks.

Transactions: A transaction is started by invoking an operation of an atomic object and terminated when the invocation terminates. AS with atomic invocations, a transaction can either commit or abort. An aborted transaction should not have any "semantic" effect on the global state of the system. Transaction can be nested by having a transaction invoke an atomic operation. A parent transaction does not terminate until all of its siblings have terminated. Failures of transactions automatically propagate down the tree (if a parent fails, all of its siblings fail) but not upwards. An atomic invocation can be designated as a **top level** transaction which has the effect of disconnecting this transaction from its parent and viewing it as being independent (i.e. can succeed or fail regardless of its creator).

Locks: Locks are used to synchronize concurrent accesses to shared resources. A lock is characterized by two attributes: exclusive/shared and atomic/aon-atomic. Non-atomic locks are explicitly acquired and released by the program. When full serialization is required, atomic locks are used. The program acquires atomic locks as required but does not release them explicitly. Instead, atomic locks are automatically released at the end of the transaction. The **Revoke** operation is defined on locks, in order to allow critical activities to take away locks from less critical ones. Revoking an atomic lock aborts the transaction holding it while revoking a non-atomic locks merely delays the transaction until the lock is reacquired (the programmer is responsible for the consistency of the system state). Revokable locks support pre-scheduling by allowing an activity to pre-acquire a lock that can be later revoked. They also provide a way of detecting failures; if an activity is using a device (e.g. robot arm) it is supposed to be holding a lock on that device. If the device fails, the lock is revoked and as a result, the activity will be affected (aborted if the lock was atomic and delayed until the device is fixed otherwise).

References

[1] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe.
The Eden System: A Technical Review.
IEEE Trans. Softw. Eng. SE-11(1):43-58, Jan., 1985.

[2] George Cox, William M. Corwin, Konrad K. Lai, and Fred J. Pollack.
A Unified Model and Implementation for Interprocess Communication in a Multiprocessor Environment.
In *Proceedings of the 8th Symposium on Operating System Principles, Asilomar*, pages 44-53. Assoc. Comput. Mach., Dec., 1981.

[3] Barbara Liskov and Robert Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Trans. on Prog. Lang. and Systems. 5(3):381-404, July, 1983.

[4] D.L. Parnas.
On the Criteria to be Used in Decomposing Systems into Modules.
Comm. of the Assoc. Comput. Mach. 15(12):1053-1058, 1972.

[5] K. Schwan, R. Ramnath, S. Sarkar, and S. Vasudevan.
Cool - Language Constructs for Constructing and Tuning Parallel Programs.
In *International Conference on Computer Languages, Miami Beach, Florida*, pages 90-103. IEEE, Oct., 1986.
Also available as technical report, Department of Computer and Information Science, Ohio State University, OSU-CISRC-86TR2KS.

[6] Tokuda, H.
Compensatable Atomic Objects in Object-oriented Operating Sysytems.
In *Proc. of Pacific computer communication Symposium.* October, 1985.

[7] K. Schwan, P. Gopinath, and Win Bo.
CHAOS-Kernel Support for Objects in the Real-Time Domain .
IEEE Transactions on Computers C-36(8):904-916, August , 1987.

Task Management Techniques for Enforcing ED Scheduling on Periodic Task Set

Aloysius K. Mok†
Department of Computer Science
University of Texas at Austin
Austin, TX 78712

Extended Abstract

Introduction

A well known algorithm for enforcing hard deadline constraints on tasks is the ED (Earliest Deadline) policy. In this policy, the task with the nearest deadline is always selected for execution whenever a scheduling decision is made. A classical result in [Liu & Layland 73] showed that for the case of independent periodic tasks, the ED policy can achieve 100% CPU utilization on a single processor without missing any deadline. (In the *independent periodic tasks model*, every task is parameterized by an ordered pair (**c**, **p**) where **c**, and **p** are respectively the computation time and period of the task, and every task can preempt any other task at any time. The CPU utilization is the sum of the ratios **c/p** of the tasks.) In [Mok 84], the ED policy was extended and shown to be optimal for the case where both task synchonization (in the style of the **ADA**‡ *rendezvous*) and mutual exclusion (under certain restrictions) are allowed. In [Mok et al 87], another variation of the ED policy was shown to be optimal also for a dataflow model with data-driven timing constraints. Hence, there is practical interest in the efficient implementation of the ED policy. For periodic tasks, however, the method of using a heap (e.g., see [Aho, Hopcroft & Ullman 74]) to organize task control blocks will in the worst case require O(n) operations to select the task with the nearest deadline where n is the number of tasks. In this note, we shall describe a O(log n) solution to the task management problem.

The Problem and a O(n) Solution

For the purpose of enforcing the ED policy, there are two parameters that the operating system keeps track of in a task control block: the current deadline **d** and the ready time **r** (the start of the current period). We say that a task is not ready if it has completed its execution for the current period and the next period has not yet started. Suppose the task control blocks are organized and stored in a data structure D. The relevant task management operations for ED scheduling are:

SELECT (D) :-	Return the ready task with the smallest **d**.
POST (π,D) :-	Insert the task π which has been preempted or which has just completed execution into D. In the latter case, set

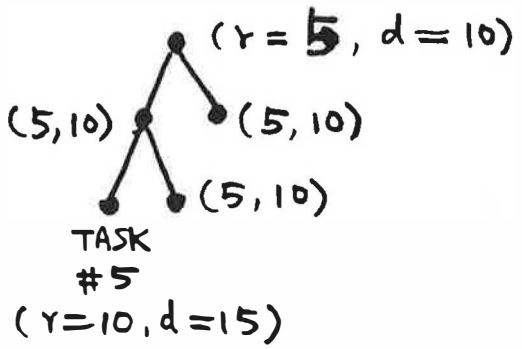
† Supported by ONR under contract number N00014-85-K-0117, by a University Research Initiative grant under contract number N00014-86-K-0763, and by a DoD University Research Instrumentation Program under contract number N00014-86-G-0199.

‡ **ADA** is a trademark of the United States Department of Defense.

d to be the end of the next period and set **r** to be the start of the next period.

WAKEUP (t,D) :- Classify any task in D as ready whose ready time **r** = t.

Since the ED policy requires a dynamic priority scheduler, a "textbook solution" is to use a heap (priority queue, e.g., see [Aho, Hopcroft & Ullman 74]) to implement the data structure D. This solution, however, has a O(n) worst-case time complexity. Consider the following periodic task set which has 5 tasks. Four of them have the same period **p**=5 and computation time **c**=1. Task #5 has parameters **p**=15, **c**=2. The first four tasks will be executed according to the ED policy after about 4 time units. At this point, the heap will look like:



When SELECT(D) is performed at around time=4, the only ready task (#5) is at the rightmost leaf on the lowest level. Thus in the worst case, the whole heap will have to be searched before SELECT(D) can find the ready task with the smallest deadline, a O(n) operation. For large n, this overhead can be unacceptable. In the above scenario, for example, it might be after time=5 before SELECT(D) can return task #5 to be executed next. By then, however, the first four tasks will have become ready again.

It should be noticed that having two separate heaps for ready and not ready tasks will not help in reducing the worst-case time complexity, since in the worst case, all of the tasks in the not-ready heap may become ready at the same time, and O(n) operations are required to move all of them to the ready heap.

The HOH (Heap-of-Heaps) Data Structure

We shall present a O(log n) solution to the task management problem. In this solution, we make use of a data structure which is a heap of heaps (HOH). A HOH is a heap whose elements can be heaps themselves. Like a heap, a HOH has the heap property: the priority of any element cannot be greater than that of any of its ancestors. A HOH supports priority queue operations:

INSERT (x,H) :- Insert x into the HOH H; x may be a simple node or a heap with two or more nodes.

DELETEMIN (H) :- Remove and return from H the node with the highest priority (smallest integer). Reorganize H so that it remains a HOH.

HOH operations are implemented very much like heap operations and are described

below.

INSERT (x,H):

Place x as far left as possible on the lowest level of H, starting a new level if necessary. If x has a higher priority (smaller integer) than its parent, exchange it with its parent. When comparing priority, use the priority of of the root if the element under comparison is a heap, else use the priority of the simple node. Repeat the comparison as many times as necessary until either the root of H is reached or no exchange is needed.

DELETEMIN (H):

Suppose the root of H is a heap with two or more nodes. Call this heap h. Remove and return the root of h and rearrange h. Call this new heap h'. Push h' as far down the HOH H as it will go. When comparing priority, use the priority of the root if the element under comparison is a heap, else use the priority of the simple node.

If the root of H is a simple node, remove and return that node and rearrange H as follows: Take the rightmost element at the lowest level of H and put it at the root of H. Then push this element as far down H as possible.

Theorem 1: Both INSERT and DELETEMIN operations on a HOH have O(log n) worst-case time complexity where n is the number simple nodes.

O(log n) Solution to the Task Management Problem

This solution uses a HOH to store ready tasks and a 2-3 tree (see, e.g., [Aho, Hopcroft & Ullman 74]) to store the not ready tasks. Each ready task is stored as a simple node in the HOH. The leaves of the 2-3 tree are heaps and each not ready task is stored as a node in one of the leaves. All the tasks in the same leaf (heap) have the same ready time (the parameter **r** in the task control block). A leaf in the 2-3 tree is accessed by using the correponding ready time of the tasks in it as the key. The data structure D for task management is a pair (H, T) where H is a HOH and T is a 2-3 tree. The relevant operations are implemented as follows.

SELECT (D) :-

Perform a DELETEMIN on H. The task returned is to be executed next.

POST (π,D) :-

If π has just been preempted, perform INSERT(π,H). If π has just completed execution, update its parameters by setting **d** to the end of the next period and **r** to the start of the next period. Locate the leaf in T which has the same ready time **r** and insert π into this heap, using the deadline **d** as the key. If T does not contain a leaf with the same ready time as π, create a heap with π as its only node and insert this heap into T.

WAKEUP (t,D) :-

Locate a leaf in T with ready time = t. If one exists (call it h), delete h from T and

insert h into H by performing INSERT(h,H).

Theorem 2: The task management operations SELECT, POST, WAKEUP can be implemented with a O(log n) worst-case time complexity where n is the number of tasks.

Representation of Clock Values

The data strture operations discussed above involve mostly arithmetic operations on time values. It is important for efficiency reasons to be able to represent time values as integers. For a 32-bit computer, a 1 millisecond basic time unit allows about 50 days before an arithmetic overflow occurs as a result of clock update. If 50 days is not a sufficiently long interval or if the mission length of an application is unknown, the following scheme may be used for representing clock values with only finite variables. The applicability of this scheme is predicated on the assumption that the values of the time parameters **d** (current deadline) and **r** (ready time) are bounded by the maximum of the periods of the tasks. This assumption holds when for example, the CPU utilization for the periodic task set does not exceed 1 in the *independent periodic tasks model.*

In this scheme, all time values are stored in finite counters (e.g., integer variables) and all arithmetic operations are performed modulo the size of the counter, say m. We use a variable: **fc** (finite clock) to keep track of the passage of time. **fc** is initialized to 0 and is incremented (mod m) at every clock tick (timer interrupt). Suppose **r** (ready time parameter in task control block) and **d** (deadline parameter) of a task which has just completed execution are respectively t and t+p (mod m) where p is its period. Then **r** and **d** will be updated to t+p (mod m) and t+2p (mod m) respectively. It is easy to see that the ready time and deadline of the task are actually **r-fc** (mod m) and **d-fc** (mod m) respectively from the current moment. These values can then be used in comparing deadlines or ready times among tasks. Thus this scheme poses no limit on how long the real-time application can run without an infinite range calendar clock.

Conclusion

In enforcing the ED scheduling discipline on periodic tasks, the straightforward way to organize task control blocks by means of a heap will incur a worst-case time complexity of O(n) per operation. We showed that a data structure with O(log n) worst-case time complexity can be used to solve the task management problem. An interesting question is whether inexpensive hardware can be built to perform ED scheduling in constant time for large but not arbitrarily large task sets.

Bibliography

[Liu & Layland 73] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming Systems in a Hard-Real-Time Environment", *JACM 20, January, 1973, pp. 46-61.*

[Mok 84] A. K. Mok, "The Design of Real-Time Programming Systems Based on Process Models", *Proceedings of the IEEE Real-Time Systems Symposium,*

December, 1984, pp 5-17.

[Mok et al 87] A. K. Mok, P. Amerasinghe, M. Chen, S. Sutanthavibul and K. Tantisirivat, "Synthesis of a Real-Time Message Processing System with Data-driven Timing Constraints", *Proceedings of the IEEE Real-Time Systems Symposium, December, 1987, pp. 133-143.*

[Aho, Hopcroft & Ullman 74] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Weslesy Publishing Co., 1974.

SESSION III

REAL-TIME OPERATING SYSTEMS

Chairman

Horst F. Wedde
Wayne State University

Co-Resident Operating System: UNIX and Real-Time Distributed Processing

by
John R. Barr, Ph.D.
Motorola Computer X, Inc.
Schaumburg, Illinois

Introduction

The co-resident operating system combines the standard operating system environment provided by the UNIX[1] operating system with a real-time distributed operating system (cXOS[2] [BaK87, BKW87, Bar87]). A key component of the co-resident operating system is the real-time kernel designed to support both operating systems. By combining the CPU dependent functions into a common kernel and implementing each operating system using this kernel, we have made it possible to combine time critical real-time applications with standard UNIX applications without compromising the performance of either environment. In addition, the Single Virtual Machine (cXSVM) process/message model that has been included in the UNIX operating system (cXV/RT) allows access to other CPUs on the same VMEbus backplane or to other nodes connected by a real-time LAN (cXLAN) that may be dedicated to a particular real-time processing task.

UNIX operating systems have been traditionally known as time-sharing systems and not as real-time systems. Recently, pressure has been applied by major users of computer systems to have a standard real-time computing system based on one of the most popular operating systems, UNIX. Some real-time UNIX operating systems are available, but the definition of a standard real-time UNIX operating system is several years away (IEEE P1003.4 Draft 1, December 7, 1987). Even when a standard real-time UNIX operating system has been defined, the real-time performance characteristics of that system will not be able to compare with the performance of real-time executives and other operating systems designed for real-time. In addition, a real-time, distributed operating system offers other advantages that a real-time UNIX operating system will not be able to offer.

---

[1] UNIX is a trademark of AT&T

[2] cXOS, cXV/RT, cXSVM, and cXLAN are trademarks of Motorola Computer X, Inc.

The Real-Time Kernel

One of the key components to any real-time operating system is the efficiency of the primary functions required for rapid response times: Process scheduling, interrupt handling, memory management, and interprocess communication. Current UNIX operating systems are not always optimized for these primary functions. In many cases, the primary goal of a UNIX operating system has been excellent interactive response to many users at terminals since UNIX was originally designed to be a program development system. The real-time kernel implemented for the co-resident operating system has several optimizations for the primary real-time functions.

Process Scheduling

Process scheduling in a multi-tasking real-time system requires some form of preemptive priority scheduling. The IEEE P1003.4 committee has agreed that there are at least two types of priority based scheduling that need to be provided in a real-time UNIX operating system: Run-to-block (SCHED_FIFO) and some other algorithm such as normal UNIX time sharing (SCHED_OTHER). There has also been some discussion about a third algorithm that would support time-slicing of processes running at the same priority level (SCHED_SHARE). Run-to-block is required for the extremely high priority processes that require dedicated use of the CPU from the time they are scheduled for execution until they have finished processing or block waiting for some other event to be completed before they continue. Time slicing is required for environments where several processes at the same priority level need to evenly share the CPU during a processing step so that they all make equal progress towards a common goal. Normal UNIX time sharing is used for background or development tasks.

Our real-time kernel implements a simple scheduling method that allows the use of all three scheduling "classes" on a per process basis. Each process is assigned a priority level, a time quantum, and a priority degradation value. At each processor scheduling, the highest priority process will be run. If that process has a non-zero time quantum value, it will be preempted at the end of that time quantum to allow another process of the same priority a "time slice". If the time quantum value is zero, the process will be allowed to run until it blocks or is preempted by a higher priority process. If a process has a non-zero priority degradation value, after it obtains the designated number of time slices defined by the time quantum value, the actual priority of the process will be lowered. If the priority degradation value is zero, the process will remain at the same priority level. Processes that are using the priority degrading policy will have their priority levels adjusted upwards, never exceeding the initial starting priority level, whenever they complete interactive system calls (signals, I/O completions, etc.) that caused the process

to block waiting for system resources. This allows processes that may have been waiting an indeterminate amount of time to be quickly run to catch up with the user. If there was no blocking in the system call, the system call is just treated as an extension of the processes time quantum.

This simple scheduling method requires minimal overhead at each processor scheduling step, but provides all three of the scheduling classes. In addition, since the real-time kernel supports both cXV/RT and cXOS operating systems in the co-resident environment, the process scheduling of cXV/RT processes and cXOS processes is done by a single CPU process scheduler. This allows for an efficient overall implementation and also allows time critical applications to execute as cXOS processes or cXV/RT processes. Applications ported from other UNIX operating systems can be quickly integrated into a high-performance real-time distributed processing environment.

Interrupt Handling

Interrupt handling in a real-time system is one of the critical performance areas. In a co-resident operating system where there is more than one operating system capable of using the interrupt handling services, a method of connecting to individual interrupt vectors is required. The real-time kernel serves as the first level interrupt handler in the co-resident system and allows driver processes (cXOS) or the interrupt handlers in the kernel (cXV/RT) to selectively "connect" to interrupt vectors. A simple kernel level dispatch table is used to get to the designated interrupt handler. In addition, the real-time kernel saves the C scratch registers and simulates a C function call that includes, as parameters, the processor fault frame (when pertinent) and a user supplied parameter that can be used to distinguish between multiple interrupt vectors using the same interrupt handler. This simple interface allows "driver" processes that include interrupt handlers to be dynamically loaded as non-resident (not included in the primary boot module) processes.

Memory Management

The memory management portion of the real-time kernel was designed to efficiently support the normal virtual memory requirements of the UNIX operating system and the requirements associated with a high-performance real-time system. Since the real-time system requirements also helped to improve the performance of several UNIX operating system functions, the new memory management design had dual purposes. The real-time requirements included efficient utilization of resources (memory space and processor time), efficient support for message passing, support for kernel virtual memory regions that could be paged, and fast fault handling in a potentially sparse virtual address space of a process.

The real-time kernel provides a function call interface to the "guest" operating system kernels implemented using the real-time kernel. The memory management portion of the real-time kernel handles the physical and virtual memory allocation for the operating system kernels. It does not handle paging to a backing store, but does provide the necessary interfaces and fault handling so that paging could be implemented in an operating system kernel. In the co-resident operating system, the cXV/RT kernel implements paging for both cXV/RT regions and cXOS regions. The implementation also allows regions maintained inside of the cXV/RT kernel to be paged.

Some of the features of memory management that have been used in the implementation of the co-resident operating system include dynamic allocation of small memory fragments (32 bytes - 2048 bytes) within the UNIX kernel; pageable kernel virtual segments for pipes, ram disk, and file system bit-maps; and the dynamic allocation of kernel data structures. The dynamic allocation of small memory fragments improves the memory usage of the kernel while reducing the amount of time it takes to obtain memory within the kernel. The pageable kernel virtual memory segments allows the creation of data structures that can easily grow and that can be treated as a simple array rather than as a set of pages. The pipe implementation is significantly more efficient than the inode oriented implementation found in most UNIX kernels. Treating the bit-map of available blocks on our high-performance file system as a single array of bytes in virtual address space results in efficient algorithms that scale well to available physical memory. The dynamic allocation of kernel data structures allows a common kernel to be used in small or large configurations without memory performance penalties.

One of our major concerns with the implementation of the process/message model in a virtual memory system was the efficiency associated with passing messages between processes. Messages are treated as memory segments that can be transferred from one process to another without copying the data. This allows for a uniform message transmission overhead that is largely independent of the message size. In addition, because messages come and go in a processes virtual address space it is possible to have a sparsely populated virtual address space. One of the design requirements for memory management emphasized this, and the result is an efficient method of mapping any virtual address into the corresponding region that contains that address. This also carries over to the use of dynamic allocation of kernel data structures and the ability to efficiently handle memory faults while in the kernel.

Interprocess Communication

Interprocess communication in a distributed environment is handled by the cXSVM process/message model. Interprocess communication local to a CPU can be handled by the cXSVM process/message model or by pipes, shared memory, and semaphores as defined by the SVID and IEEE POSIX P1003.1 and P1003.4. In either case, the efficient handling of memory segments that can be attached and detached from a process is necessary for optimal performance. Messages passed between processes local to a CPU is handled by removing a memory segment from the sending process and adding that memory segment to the destination process. Since both cXV/RT and cXOS utilize the same memory management scheme on the co-resident system, messages can be passed between processes without regard for which operating system is handling the processes.

When the process/message model is used for interprocess communication, processes may be allocated to other CPUs on the VMEbus or to CPUs in another node on the cXLAN without changing the application software. This allows applications to be written that are independent of the underlying hardware configuration. Using the cXSVM process/message model, it is possible to write applications that can be scaled from small single CPU implementations that are cost sensitive up through many nodes on a LAN where each node may contain multiple CPUs. Resources such as disks, operator consoles, device interfaces, etc. may be added or deleted to such implementations without changing application software.

Shared memory is the highest performance method of interprocess communication available when it is known that the communicating processes will share a common global memory address space. This restricts an application implementation to a single node in a distributed environment and requires careful coordination between cooperating processes to ensure that data structures are not corrupted by allowing multiple processes to modify data at the same time. Semaphores or selective write access can be used in such cases. The memory management services provided by the real-time kernel support concurrent access to common memory segments with selective read/write access.

Programming Environment

The co-resident operating system provides a programming environment that includes all of the features of a standard System V UNIX operating system and those features unique to a real-time distributed operating system that was designed specifically for applications such as factory automation. The programming environment allows an applications programmer to write applications that include the cooperation of UNIX processes and processes

implemented for cXOS that may be distributed amongst other processors in the cXSVM (Figure 1).

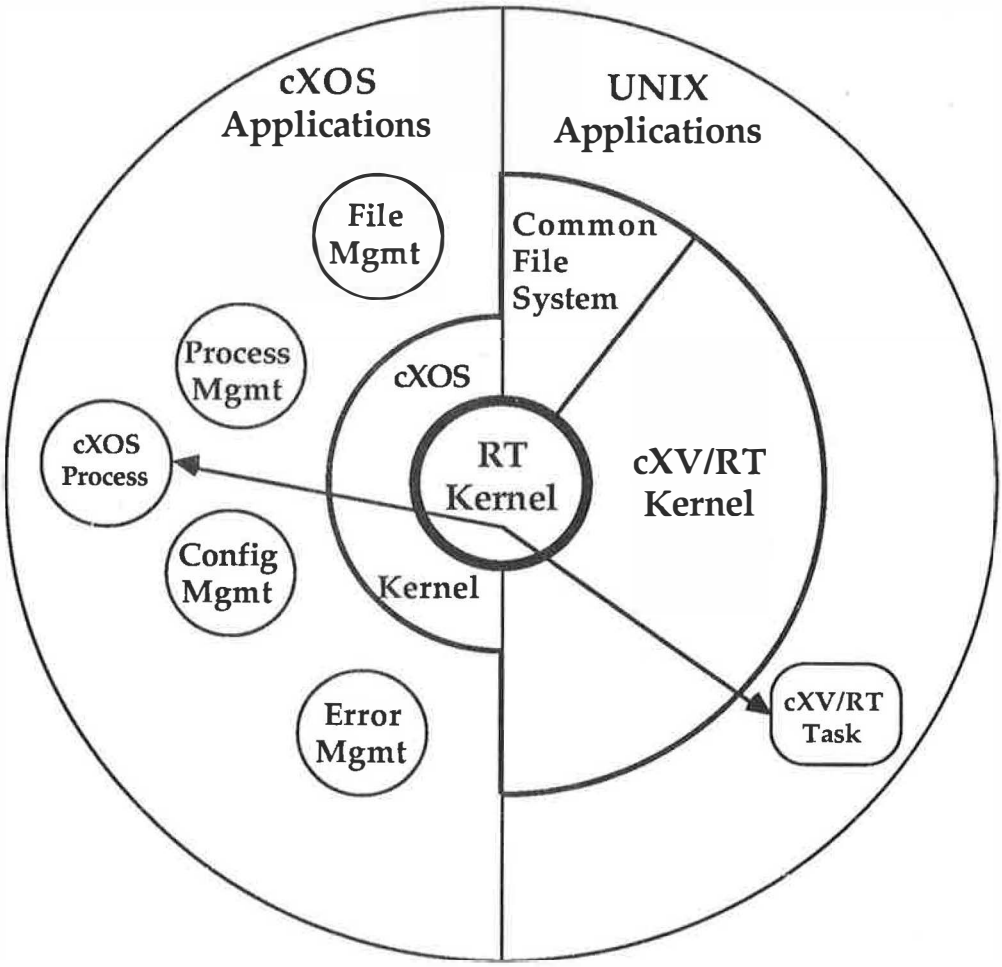


Figure 1: cXV/RT - cXOS Co-Resident Operating System Environment

Comparison with Other Systems

One of the other UNIX operating systems that has been designed for distributed processing is the Mach effort at CMU [Ras86, Tev87]. Even though Mach has not been designed for real-time performance, some of the features of Mach required to support distributed multi-processing also can be found in our real-time kernel. We differ in the method of message passing since Mach implements a copy oriented view using copy-on-write mapping instead of removing the memory space from the sending process and adding it to the destination process. Mach also implements extensions to the UNIX operating system to support their version of message passing to allow processes on

different nodes connected by a LAN to efficiently exchange information. The "threads" provided by Mach are both a feature and an "opportunity". Threads work fine in an environment where there are no asynchronous scheduling paths that are associated with the "task" containing the threads. Tevanian emphasizes that the use of threads within a normal UNIX process is not yet a well defined mechanism. We have chosen to make the creation and maintenance of processes low overhead operations so that one can merely use the normal UNIX or cXOS process model instead of forcing programmers to implement their own process model inside of a normal process.

Conclusions

The co-resident operating system provides some unique benefits for application programmers that must cope with a real-time distributed environment. The combination of a real-time UNIX operating system and a real-time distributed operating system on the same CPU improves the cost effectiveness of implementations that require applications to span both systems. The cXSVM process/message model has been efficiently implemented in a real-time kernel used to implement such a co-resident operating system. With the combination of cXOS and cXV/RT on the same CPU, application programmers can take advantage of the best services of each environment without having to suffer any penalties.

References

BaK87 Barr, J. R. and Andrew Kun, *The Computer X Distributed, Real-time System*, July, 1987, Presented to the IEEE Fourth Workshop on Real-Time Operating Systems, Cambridge, MA, pp. 55-58.

BKW87 Barr, J.R., Andrew Kun, and Bernhard Weisshaar, *Computer X, Real-Time Distributed Operating System,* Presented to the IEEE International Conference on Computer Design, October, 1987, Rye Brook, NY.

Bar87 Barr, J. R., *A Real-Time Distributed Operating System*, Presented to the 17th International Symposium on Automotive Technology and Automation, October, 1987, Munich, Germany.

Ras86 Rashid, R., et al, "Mach: A New Kernel Foundation for UNIX Development", *USENIX Conference Proceedings*, Summer, 1986, Atlanta, Georgia, pp. 93-112.

Tev87 Tevanian, Avadis, et al, "Mach Threads and the UNIX Kernel: The Battle for Control", *USENIX Conference Proceedings*, Summer, 1987, Phoenix, Arizona, pp. 185-197.

The SAGE Operating System

Lou Salkind
Robotics Group
New York University

February 29, 1988

1 Introduction

In robotics and other manufacturing applications, real-time supervisory control is often required to coordinate several different sensors and actuators. In many cases, supervisory controllers must also interface to other computers which perform higher-level tasks such as motion planning or factory-level coordination. Therefore, the supervisory controller must not only interface to a variety of physical devices, but it must also support a variety of communication disciplines.

SAGE is an operating system designed specifically for developing and supporting real-time supervisory control applications. The system provides a number of flexible interfaces and debugging tools, which reduces the amount of time spent interfacing to external devices and computers. In addition, the system was designed so that new devices and protocols can be quickly incorporated into the system.

In many respects, SAGE is similar to more conventional operating systems, since it provides multi-tasking, memory management, communications, and a number of supported devices. What is unusual about SAGE is that all these facilities can be provided while still guaranteeing real-time response to supervisory control processes.

2 System Architecture

A simple architecture was adopted for SAGE. SAGE programs are developed and cross-compiled on a UNIX workstation, and then dynamically downloaded (over either an ethernet or serial line) to the SAGE host, which consists of a resident kernel running on a 68000 processor board. SAGE programs invoke operating system services by trapping to the executive.

At the system call level, SAGE presents an interface somewhat similar to UNIX, and in fact, a large number of UNIX system calls are either directly provided or can be emulated in SAGE. However, UNIX compatibility was only maintained where convenient, and many of the facilities such as process control and creation are radically different.

The SAGE kernel, as well as most of the support libraries, are written in C. SAGE applications are also generally written in either C or C++. The system supports most of the standard C library routines, including standard I/O, memory allocation routines, and math functions for a number of floating point co-processors. SAGE also supports programs written in Fortran.

By relegating program development functions to the UNIX host, the SAGE kernel and support software can be kept relatively simple. Only those time critical operations necessary for supervisory control are implemented in the SAGE kernel. Other generally useful facilities, such as file access, are implemented as remote procedure calls made to a UNIX server process.

3 SAGE Features

We briefly overview some of the major components of the SAGE kernel, and describe why these features were necessary in our supervisory control applications.

3.1 Multitasking

Because a supervisory control system must typically control several devices and communication streams simultaneously, each with relatively low (but unpredictable) duty cycle, SAGE was designed as a multitasking system. This allows an application to dedicate a task (or several tasks) to each external device or stream.

There are two types of tasks in SAGE:

- a (full-weight) process, which has its own stack and memory management context, and can invoke the full range of system services. In particular, a process can be blocked, suspended, or preempted.
- an interrupt handler, which runs on the current process's stack, can access any valid address, and can only invoke a limited number of operations. Because it shares the process's stack, an interrupt handler cannot block, and it can only be preempted by a higher priority interrupt handler.

In general, full-weight processes are appropriate for supervisory control, because the event driven nature of the applications imply that a process may be suspended and resumed at unpredictable times. Interrupt handlers are used in those cases where microsecond level response is required, or an unsupported device needs to be interfaced to the system.

An important goal in SAGE was to allow full-weight processes to meet millisecond level time constraints. To achieve this, the kernel was structured to be preemptible in most places. In addition, every effort was made to shorten those sections of the kernel that were not preemptible. In particular, the system-supplied interrupt handlers, which cannot be preempted by regular processes, were written so that they ran for only a limited amount of time. Any lengthy operations required by a handler were queued for later execution by a preemptible kernel-resident process.

Process priorities range from 0 to 31, with 31 the highest. By default, SAGE processes execute at the processor's base priority, so all interrupt handlers effectively have a higher priority than any process. However, a SAGE process can elevate its processor priority as needed to mask out interrupts.

The SAGE scheduler always runs the highest priority ready process, and it will not adjust process priorities dynamically. In particular, the scheduler does not provide other policies such as round-robin preemption or earliest deadline first. There are hooks, however, for an intelligent process to control the priorities of other processes, and thus simulate particular scheduling policies.

3.2 Memory Management

Many real-time systems avoid using memory management features because of the (supposedly) high overhead. SAGE, however, views memory management as an essential component of a supervisory control system, since it provides both protection and fault containment in a multitasking environment. In addition, SAGE also exploits memory management hardware to improve the overall performance and functionality of the system.

SAGE memory management operations center around the notion of a segment, which is simply a range of virtual addresses. Segments are named objects, and they can be mapped into a process's address space by using an open system call. Shared memory is provided by having several processes open the same segment simultaneously. Shared memory is considered essential for real-time applications because of the high bandwidth it provides for interprocess communication.

Although a segment's virtual addresses are generally mapped into a set of physical pages, SAGE also provides a control operation that allows a segment's addresses to be mapped into either bus memory or I/O space. Another control operation allows a segment's data to be mapped out onto the bus, where it can be accessed by DMA bus masters. Thus the system provides a general memory mapped I/O capability. Furthermore, since users can define their own interrupt handlers, it is very easy for applications to supply their own device drivers.

Another control operation allows the process to set segment protections such as read-only, read-write, etc. The typical process will create at least three segments for itself: a read-only text segment,

a read-write data segment, and a read-write stack segment. Segment protection, combined with the ability to invalidate individual pages, allows a SAGE process to set firewalls in its address space.

All SAGE segments currently allocate virtual memory from the same pool. In general this is quite safe, since part of the process's context is a list of mapped-in segments. The system makes sure that a process can only access its mapped in segments and no other addresses.

Page mapping is also used by SAGE to avoid copying data in and out of the kernel. To do this, SAGE provides special versions of the read and write system calls, **pgswapread** and **pgswapwrite**, that swap pages with the kernel instead of copying data between address spaces.

Since neither of these calls preserve copy semantics, they are much simpler to implement than the "lazy evaluation" techniques used in Accent [2]. However, because the SAGE calls are destructive, they are somewhat less convenient to use than the Accent equivalents. Nevertheless, the SAGE calls have proved useful for real-time work, since copying can still be avoided in most cases, and page faults are guaranteed not be taken at inappropriate times.

3.3 Synchronization Primitives

SAGE provides two basic facilities for process synchronization: a locking facility, whereby processes can temporarily become non-preemptible, and a scheduling facility, whereby processes can suspend and resume themselves. The facilities are intended to allow application dependent synchronization primitives to be built with a minimum of system overhead. Variations of these calls were first proposed in [1].

The locking primitives are used to insure that a process, once inside a critical section, cannot be preempted. This allows atomic operations to be constructed, from which more sophisticated synchronization primitives can be built.

Essentially, two words (**word1** and **word2**) are shared between the process and the scheduler. **word1**, when set by the process, informs the scheduler that the process should not be preempted. Once the process has finished its critical section, it clears **word1** again, which allows the scheduler to preempt the process.

word2 is set by the scheduler when it tries to reschedule a process but cannot because **word1** is set. The understanding is that the process will, after clearing **word1**, check **word2**. If **word2** is set, the process will request that the CPU be rescheduled.

In the usual case, rescheduling is not required, and only a few instructions are required to lock and unlock a critical section. In particular, system calls are avoided. This is important if shared memory is to be used effectively between tightly coupled processes

The other essential building block for synchronization operations is a mechanism to suspend and resume processes. SAGE provides the primitives:

suspendproc(pid, flag)	suspend process **pid** if **flag != 0** or **flag == 0** and the RESUME_CALLED flag has not been set for process **pid**. In any case, clear the RESUME_CALLED flag for **pid**.
resumeproc(pid)	resume **pid** if it ps suspended; otherwise, set the RESUME_CALLED flag for **pid**

The RESUME_CALLED flag associated with **suspendproc** and **resumeproc** allows these primitives to be called in either order with the same effect, thus avoiding potential race conditions.

Using the locking and process scheduling primitives, a wide range of user-level synchronization primitives can be constructed. Currently implemented primitives include binary and counting semaphores, as well as the UNIX style sleep and wakeup. Again, this code is quite fast in the usual non-blocking case, since system calls are avoided.

3.4 Timing Facilities

SAGE provides both relative and absolute timers, the main ones being:

pauseabsolute	pause a process until a certain time
pauserelative	pause a process for a given amount of time
alarmabsolute	interrupt the process at a certain time
alarmrelative	interrupt the process after a given amount of time
gettimeofday	return the current time
settimeofday	set the current time

In addition, a few profiling and instrumentation timers are provided.

Note that if only relative timers were provided, there would be no way to insure that a timer event is generated at a particular time, since the process could not atomically get the current time and calculate the appropriate interval.

All time arguments are specified to microsecond resolution, since high resolution timers are needed to avoid quantization errors. In the current implementation, however, the effective resolution of the time of day and interrupt timers is limited by the hardware to roughly 5 microseconds. Furthermore, because the number of hardware timers is limited, profiling timers are performed by a sampling routine called at a configuration-dependent frequency, which is usually set at 50 HZ. Therefore, the profiling timers still have very coarse resolution.

3.5 Distributed Communications

Supervisory control systems work in an inherently distributed environment, so it is important that the system supports network communications. SAGE was designed to efficiently support multiple protocols running on different physical networks.

Multiple protocols need to be supported for several reasons:

- No single protocol seems appropriate for all real-time applications, because of the inherent tradeoff between the reliability and throughput of the protocol.
- Certain hosts have not implemented certain protocols. For an application to communicate with these hosts, it is generally easier to support the host's protocol in SAGE than to modify the host's software.

Currently, several of the standard Internet protocols have been implemented, including IP and UDP. In addition, a reliable datagram protocol (built on top of UDP) has been implemented for remote procedure calls.

For all the protocols, the kernel takes care of low-level network details such as routing, checksums, fragmentation and assembly, etc. The user is presented with a simple open/close/read/write interface to the network.

3.6 Interprocess Communication

Interprocess communication can be accomplished through either shared memory or message passing. Currently, a message passing stream provides a single full duplex channel between a pair of cooperating processes. In the future we plan to provide a multiplexing capability so that multiple senders and receivers can use a message stream.

3.7 Debugging Facilities

The system has well defined interfaces for use by debuggers. For instance, a process can specify the location of its symbol table, which allows debuggers to dynamically attach to processes. There are also facilities to peek and poke a process's address space, trap exceptions, set breakpoints, etc.

Several debuggers for the system exist, including both a kernel and an application level debugger. In addition, there is an error logger (implemented as an application process), which is used for capturing data during experiments.

Function	SAGE	SUN 3/160 (3.2)
System Call	50-80 usecs	120 usecs
Null Process	1 msec	12 msec
UDP 4K writes	367K bytes/sec	374K bytes/sec
Context Switch	60 usecs	

Table 1: SAGE vs. UNIX

4 Current Status

SAGE is now running on several different Pacific Microsystems 68000-based processor boards, including both Multibus and VMEbus systems. The essential hardware components used by SAGE include the memory management unit (for protection and memory mapped I/O), dual ported memory, two high resolution timers, and on-board software interrupts. These components are found in most minicomputers and many microcomputers.

Extensive benchmarks have yet to be performed for the SAGE system. However, preliminary figures illustrated in Table 1 show SAGE's performance is competitive with UNIX systems for common functions. Here both SAGE and the SUN are using 68020 processors clocked at 16.7MHz.

Of course, the benchmarks should not be taken too seriously, since the systems are totally different in many respects. Indeed, SAGE's only performance goal has been to support the type of supervisory control applications performed in our laboratory. In this regard, SAGE has been successful. In particular, one 68020-based system simultaneously handles several thousand interrupts a second and a number of active network connections, all while still providing real-time response on the order of milliseconds.

In the future, we hope to port SAGE to several other types of processor boards and to build a multiprocessor system. We would also like the extend the kernel in several ways, including adding the ability to handle signals and exceptions in a timely fashion.

References

[1] J. Edler, A. Gottlieb, J. Lipkis, *Considerations for Massively Parallel UNIX Systems on the NYU Ultracomputer and IBM RP3*, USENIX Winter Conference Proceedings, pp. 193-210, January 1986

[2] R. P. Fitzgerald, R. F. Rashid, *The Integration of Virtual Memory Management and Interprocess Communication in Accent*, ACM Transactions on Computer Systems, 4(2), May 1986

PANEL

POSIX

Chairman

William Corwin
Intel

Panelists

John R. Barr
Motorola

Sol Kavy
Hewlett Packard

Gregg Kellogg
NeXT Incorporated

Marc Donner
IBM Research

NOTES:

NOTES:

SESSION IV

REAL-TIME SOFTWARE DEVELOPMENT

Chairman

Robert Cook
University of Virginia

AN OVERVIEW OF

ARCHITECTURAL DIRECTIONS FOR

REAL-TIME DISTRIBUTED SYSTEMS

by

Pat H. Watson

IBM Federal Systems Division
9500 Godwin Drive
Manassas, Virginia 22110

ABSTRACT

The purpose of this paper is to provide a comprehensive real-time distributed system perspective and to use that perspective as a basis to stimulate discussions that help focus current and future research activities. The following presents a perspective on future architectural directions for complex, mission-oriented, real-time systems, as is represented in Figure 1. It identifies some key measurements of real-time architectural effectiveness as well as key hardware building blocks that require real-time latency management. Concepts are presented for process, data base, and network partitioning as well as for system-level hardware failure recovery and software fault tolerance. Finally, the key problem in real-time system management today is briefly addressed.

REAL-TIME ARCHITECTURE MEASURES OF EFFECTIVENESS

Since our focus is on real-time systems (systems with hard deadlines), the architectural Measures Of Effectiveness (MOEs) discussed here will pertain to that class of systems. Furthermore, many of these systems and much of the ongoing research is related to military applications. Our MOEs will be relevant to that environment as well. We are addressing "architectural" measures because they are somewhat independent of application, and our concern is with research and development of the system infrastructures upon which real-time applications are based. These MOEs are meant to guide design and are a basis for determining the relative merits of one approach compared to another.

For real-time systems with hard deadlines, the first architectural MOE is related to the basic architectural performance of the system. The system should predictably satisfy all of its response requirements for tasks, messages, and data base transactions, the system's basic units of work. These response requirements include periodic as well as aperiodic scheduling requirements. The architecture must also deal with concurrency constraints associated with precedence relationships and data base consistency management. For an architecture to achieve a high rating for this MOE, it must explicitly manage its shared resources, such as CPUs, disks, and Local Area Networks (LANs), with a consistent policy that assures adequate and predictable end-to-end response performance. Ideally, the performance is predictable using closed-form analysis rather than discrete event simulation. The degree of certainty associated with the prediction of meeting all response requirements is extremely high.

The second architectural MOE is related to cost effectiveness and system availability. The satisfaction of response requirements must be accomplished even when the shared resources are highly utilized (70% or greater). For many military real-time systems, such as weapons platforms, the maximum operational need for system stability and performance coincides with the system operating under its heaviest load. Stable system performance under high utilization allows an application to be supported with fewer resources, thus

reducing initial cost, weight, and number of spare parts. This makes redundancy more affordable and allows the system architect to include online redundant resources that yield higher system availability.

A third MOE is whether the architecture effectively supports process-level reconfiguration. Most of the real-time systems of interest (for example, weapons and space systems) need to operate in the presence of failed hardware resources. Many of them need to operate for long periods (some permanently) without repair. The ability to reconfigure the software on the hardware at the process level and sustain the most critical application functions as hardware assets diminish is becoming very important to many application areas. The problem is how to achieve the first two MOEs as well and avoid an insurmountable system test requirement. The tremendous number of degraded mode configurations resulting from process-level remapping of software is both the solution and the problem resulting from this method of increasing system availability. One viable approach for achieving a high rating in this category is to make the system response performance analyzable through closed-form analysis that accurately predicts whether deadlines will be missed. Such an approach will not only solve the testing problem, but will also provide the online algorithms necessary to find viable reconfiguration options in real time.

A final measure of architectural effectiveness is the timeliness of updates to real-time systems and installation of updates to operational systems. This ability is very important to military weapon systems as threat characteristics and algorithmic techniques are constantly changing. The ability to update these systems quickly has a direct bearing on their state of readiness. A key problem is the amount of system regression testing that must be performed once a change is installed. The tests must prove that there are no impacts on the performance of unchanged capabilities as a result of new demands on shared system resources. Fortunately, an architecture that scored high on the previous MOEs should easily score high on this one.

SYSTEM RESOURCES AND RESOURCE MANAGEMENT

The shared system resources previously discussed include processors, LANs, and data base management resources. The solution to the real-time system problem lies in the proper management of these resources to meet the latency requirements of the application. A totally integrated approach is required. A system that ideally manages CPUs to meet latency requirements will probably fail (miss critical deadlines) if it does not explicitly manage latency in the use of network or data base resources.

For each of these key resources, the following are architectural projections for the future. With respect to General Purpose Processors (GPPs), many future real-time systems (especially embedded military ones) will be built around one or both of the 680X0/80X86-based family of CPU chip sets. With extensive use of distributed architectures, real-time optimized (latency management-oriented) run-time environments will be developed for these two basic processor architectures in the very near future.

LANs are migrating to fiber-optic bus/ring topologies. LANs should be designed with multidomain operations in mind; that is, several homogeneous rings/buses networked together. This provides functional isolation for communications, multiplies total system bandwidth, and provides an architecture that can be easily expanded to accommodate substantial growth in system hardware assets. LANs will also have real-time optimized Network Operating Systems (NOSs) which explicitly manage message and packet communication latency. As the integrating element of the architecture, LANs should provide support for system services, such as system time and global semaphores. LANs should be designed to help solve those real-time distributed system management problems that they are well-positioned to solve, even though it broadens their area of responsibility beyond that of communications.

In the area of data base management, we can expect to see increased optical disk usage. Magnetic disks will not disappear until access performance and cost are equalized. Data base co-processors that reside on GPP control and data buses will be used extensively in future systems. Their superior performance is required to support real-time data base response requirements associated with automated decision making processes. Since the data base co-processor is on the GPP internal bus, it is available for more extensive use than if it were a remote processor.

For very simple transactions, the co-processor will be slower than a local memory-based approach; however, it will offload the GPP's CPU. For complex transactions, the benefits will be very substantial. Both memory-based Data Base Management Systems (DBMSs) and data base co-processor approaches are needed, sometimes even in the same system. Real-time data base managers will make use of latency management techniques to assure response performance and to make that performance predictable. File servers will use these techniques to manage disk accesses so that the more time-critical transactions are not blocked by less time-critical ones for more than an acceptable quantum of time (for example, one revolution of the disk).

In support of automated decision making, there will be a blending of data base, rule-based, and artificial neural system technologies. This is not an either/or situation; it is a question of applying the right tool to the right job.

In the area of programming languages, the required changes will be made to support latency management for task, message, and data base scheduling for the distributed real-time environment. However, continuing quantitative justification for the changes must be forthcoming from the research community.

SYSTEM PARTITIONING

There are several key assumptions upon which the following approach to real-time distributed system partitioning is based. The first assumption is that only message and data base management interfaces between processes are allowed. A process may share a CPU with other processes but it may not share local memory with other processes. A process is defined as containing one or more tasks, or schedulable entities, that may share memory. The use of shared memory by tasks within a process is necessary to achieve real-time performance for many applications.

The second major assumption is that disk maps are primarily established at system design time and data base ad hoc queries are the only "tasks" not known at system design time. A data base transaction can be considered as a logical task since it defines the processing to be done by the data base manager.

The above assumptions are appropriate for mission-oriented real-time distributed systems such as might be found on aircraft, surface ships, submarines and space platforms.

Processor/Process Partitioning

Processors can be categorized as dynamic or static with respect to the permanence of the processes they support. A dynamic processor by this definition contains processes that are very dependent on the current/tactical situation. As a result of a changed operating environment or upon operator request, these processors receive new process loads. It is to be expected that dynamic processors will contain shadow data bases; they may or may not contain data base tables that are static in nature.

A static processor contains process loads that are moved only as the result of hardware failures. Examples of such process loads include those associated with system-level managers, continuous background jobs, and prime data base managers.

Data Base Management Partitioning

The following is a discussion of data base management resource partitioning which addresses processor memory-based DBMSs, data base co-processors, and file servers. Potentially, each GPP would have a memory-based DBMS. The most frequently accessed tables, as well as those with demanding response requirements, would be in memory. Large tables would be stored on disk or in data base co-processor memory. Those tables which must be preserved through hardware failures will be redundantly stored, on disk or in computer memory.

The use of local Random Access Memory (RAM) for storing frequently accessed tables is critical to real-time performance for many applications. There simply is not enough time to go to a remote disk, data base co-processor, or a remote DBMS to acquire the data. This will require shadow copies of data tables at some processors. The use of a prime-then-shadow update sequence is useful for data base consistency management.

Some, if not all, GPPs should have a data base co-processor. The co-processor should reside on the GPPs data and control bus and have several megabytes of its own storage. Its architecture should be pipelined for parallel operations on multiple data fields within single records. Entire data table records should be processed at speeds that are equivalent to single data item processing in the GPP. The data base co-processor should be expected to have a mix of static and dynamic tables.

File servers located at each disk node will be responsible for control of disk accesses in accordance with latency management concepts. The file server should schedule each access on a disk-rotation basis using a prioritized access queue. This will provide a natural level of interruption that allows an access with more demanding response requirements to take precedence over those with less demanding requirements. This may cause a reduction in data throughput because of the possibility of more disk seeks, but the response performance of the system will be significantly improved and system timing behavior will become predictable. Due to relatively long disk-rotational delays, the overhead associated with disk-access scheduling will be negligible and will be accomplished in parallel with the disk-rotational delay.

One issue is whether the disk and its file server should be on a separate node or associated with a GPP. The best answer, if performance permits, is that it be a separate node so that processor failure does not have an impact on disk availability. Also, because processor loads move as a result of processor failures or tactical reconfigurations, it would not be practical to migrate disk loads with the processors for most systems. Migration is a difficult process as disks are constantly being updated and the time required to migrate far exceeds allowable reconfiguration response requirements.

The data management design should make table/file location logically transparent to the application. It will be up to the system data base managers to find the network tables/files and to issue the appropriate commands to satisfy the application needs. For mission-oriented systems, the system architect must assume responsibility for locating the data tables/files to assure that system response performance requirements are achieved.

Network Partitioning

For large, distributed, local area systems, there is a need for multidomain networking in which each domain contains a separate set of processors. The domains should be connected by bridges that store and forward packets. Additional domains may be added for redundancy where required. Not only does this multiply the bandwidth in the total network and make it easy to add whole new domains and processors, but it also provides for functional isolation of traffic. All traffic should not have to share the same bus/ring; it only increases the probability of performance problems as system requirements grow.

While physical network partitioning is good, it is very important that there be no logical partitioning. The applications should deal with a single process port to process port logical address space, no matter how many physical domains are in the network. It is also essential to be able to predict and schedule communication latency across the entire multidomain network. The use of priorities derived from latency requirements are an excellent means of scheduling network resources across domains whereas processor or node priorities would be meaningless. Priorities derived from physical location can even be counterproductive within a single domain if the processes move around as a result of hardware failures or tactical reconfigurations.

HARDWARE FAILURE RECOVERY

The reconfigurable elements of a real-time distributed system include: processors and data base co-processors, file server nodes and associated disks, LAN interfaces, and the networks themselves. While

repair activities may require fault localization below this level, it is sufficient for system reconfiguration management.

Acceptable schemes for processor/co-processor recovery include moving the total process load to a redundant processor, taking over a processor with a less important load, or dispersing the process load across the remaining processing assets. The latter option yields far higher availability but requires response performance to be analyzed online in order to find and assure a viable reconfiguration option.

When processes are relocated, the local DBMS tables must be reconstructed while the remainder of the system remains operable. This is a very challenging problem in a real-time system.

File server/disk recovery involves switching to backup tables on other disks when the failure initially occurs, rebuilding the disk image on a spare file server/disk node, and then resuming operation with the rebuilt disk. The rebuilt disk should be optimally mapped to serve the needs of its known transactions. From a system perspective, therefore, it is better to resume its use rather than staying with the backup disks.

SOFTWARE FAULT TOLERANCE

While acknowledging that this is a complex subject, a system philosophy is presented here which can be used to guide design choices. First, the requestor of a service must be responsible for taking action when the service is not rendered on time. This places responsibility for taking action and leaves the response with the designer, who is best able to determine the functional recovery action. Often it is crucial to keep certain data flow going in order to keep the system going. Frequently, the receipt of data is the event that triggers multiple processing threads. Some may not be affected at all by the failure that interrupted the data flow. When faults are detected, best-effort processing should be attempted to the extent it is feasible, not overly costly, and consistent with the seriousness of the failure. If best-effort processing is attempted, it is also necessary to provide a mechanism to deactivate malfunctioning software. For example, if more than one failure occurs in a certain time period, the executive portion of the operating system should deactivate the offending software and check to see if a backup process needs to be initiated.

THE KEY PROBLEM IN REAL-TIME SYSTEM MANAGEMENT TODAY

The key problem in real-time system management today is the lack of integrated system-wide approaches to managing task, message, and data base transaction latencies. It is not sufficient to provide latency management in processors and ignore it in the network that is the integrating element of the system. This is especially true for those systems with processing threads that span multiple processors and hard end-to-end response requirements. The shared resources which must be scheduled using time-driven techniques (such as rate-monotonic scheduling) include as a minimum: CPUs, data base co-processors, network interface units, I/O channels (virtual or real), buses/rings, and disks.

This is a fundamental problem whose solution will result in major cost and performance benefits to real-time systems. Research to develop algorithms and latency management techniques will continue, but those that have been developed should be rigorously tested on distributed system hardware testbeds.

REFERENCES

The following list of papers documents research results funded whole or in part by the Federal Systems Division of the IBM Corporation.

1. Rajkumar, Ragunathan, Lui Sha, John P. Lehoczky. "On Countering the Effects of Cycle-Stealing in a Hard Real-Time Environment." *Proceedings of the Real-Time Systems Symposium.* pp. 2-11. San Jose, California: Computer Society of the Institute of Electrical and Electronics Engineers, December 1987.

2. Tokuda, Hideyuki, James W. Wendorf, Huay-Yong Wang. "Implementation of a Time-Driven Scheduler for Real-Time Operating Systems." *Proceedings of the Real-Time Systems Symposium.* pp. 271-280. San Jose, California: Computer Society of the Institute of Electrical and Electronics Engineers, December 1987.

3. Lehoczky, John P., Lui Sha, J. K. Strosnider. "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments." *Proceedings of the Real-Time Systems Symposium.* pp. 261-270. San Jose, California: Computer Society of the Institute of Electrical and Electronics Engineers, December 1987.

4. Lehoczky, John P., and Lui Sha. *The Average Case Behavior of the Rate-Monotonic Scheduling Algorithm*. Tech Report, Department of Statistics, Carnegie-Mellon University, 1986.

5. Lehoczky, John P., and Lui Sha. *Performance of Real-Time Bus Scheduling Algorithms*. Tech Report, Department of Statistics, Department of Computer Science, Carnegie-Mellon University, 1985.

6. Jensen, E. Douglas, John P. Lehoczky, Lui Sha, Samuel Shipman, Martin McKendry. *Scheduling Hard Deadline Periodic Tasks with I/O*. Final Report, Distributed System Reconfiguration Study, Logic Associates, 1985.

7. Jensen, E. Douglas, John P. Lehoczky, Lui Sha, Samuel Shipman. *Advanced Distributed System Control Study: Reconfiguration Methodology*. Final Report, Distributed System Reconfiguration Study, Logic Associates, 1984.

8. Jensen, E. Douglas, John P. Lehoczky, Lui Sha, Samuel Shipman. *Advanced Distributed System Control Study: Reconfiguration Methodology*. Final Report, Distributed System Reconfiguration Study, Logic Associates, 1983.

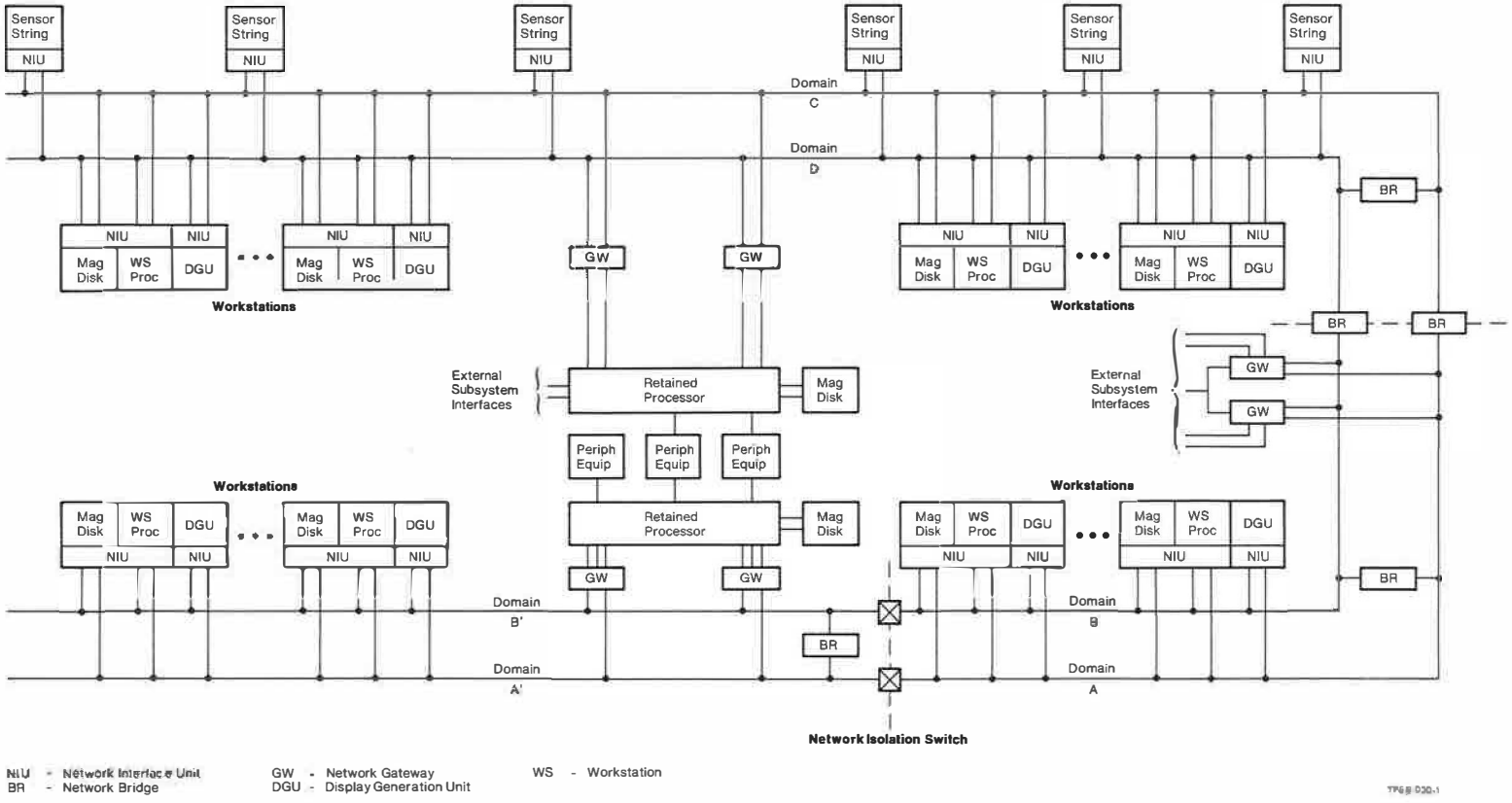


Figure 1. Local Area Distributed Real-Time System with Multiple Domain Communication Network.

DEBUGGING DISTRIBUTED REAL-TIME SOFTWARE

by
Venu P. Banda
Richard A. Volz
The Robotics Research Laboratory
The University of Michigan
Ann Arbor, Michigan 48109

ABSTRACT

Introduction

The debugging of distributed real-time programs is an activity that has very few support tools and can therefore be an extremely difficult task. Particularly difficult to find are latent errors that usually only appear after a long operating time upon the occurence some unusual combination of events. Recreating the situation or making the association between the failure and the particular occurence of events is often almost impossible. Yet, distributed real-time systems are rapidly becoming the norm rather than the exception. It is thus important that new, more effective debugging tools be developed.

While there has been a great deal of interest in debugging parallel programs recently, very few attempts have been targeted towards debugging parallel real-time software. Most debuggers interfere with the target program, thereby perturbing their behavior. One of the notable attempts, the Bell system 1A [Wits83], built by Bell laboratories for their electronic switching system application, provided for debugging through varying levels of interference. Its main drawback, however, is that one must plan the debugging session before executing the program. Plattner [Plat84] has developed a non-interfering debugger by monitoring program execution through explicit monitoring hardware attached to the system bus. A secondary processor analyses all the information collected from the bus, searching for event sequences specified by the programmer via a predicate action language. When an error is detected, the user is given control. However, this scheme suffers from the drawback that one has to know the correct event sequences before program execution. Moreover, once the programmer gets control, the debugger interferes with the execution of the program. Other notable efforts [Wile83],[LeBl85] are all some variation of the event-action paradigm of debugging. All these approaches express events at a high level, which is desirable from the user's point of view but detection of these events *non-interferingly* is a major hurdle. Using an auxillary processor will not help because trapping an action corresponding to an event would require the auxillary processor to steal some bus cycles from the target processor to access the memory, which is an interfering action. We feel, therefore, that all these approaches are inadequate for detecting timing errors.

Philosophy and Issues

Since it is clearly not possible to debug a real-time program by interacting with it while it is executing in actual operation without causing interference, we follow a two phase approach of monitoring program execution and then recreating the execution in a subsequent phase during which a programmer can use the usual kind of interfering activities, e.g., setting breakpoints, etc. Thus, the ability to monitor program execution is an activity fundamental to debugging, and support for performing this activity *without perturbing the target program execution* is absolutely essential in detecting time critical errors. We believe strongly that hardware support is essential to accomplish this. In addition, we believe that it is necessary to permanently include a small amount of code to assist debugging in every real-time program. However, such code would be non-interferring because it is permanently in the program and always executed. Obviously, the amount of such code must be kept small in comparison to the principal work being performed.

The literature is surprisingly lacking in studies of the basic issues in monitoring and reconstruction of program execution, e.g,

- What does it mean for the execution of a program to be reproducible?
- What should one monitor in order to be able to reproduce a program's execution?
- Can one find any necessary and/or sufficient criteria for a program execution to be reproducible ?

Furthermore, little has been done to develop techniques to support long term error detection i.e, detecting errors that only surface long after the program has been certified as functioning correctly.

In subsequent sections we will describe a new approach to debugging distributed real-time programs that solves the issues mentioned above.

A New Approach

As noted above, we advocate a two-phase approach to debugging parallel programs. Phase I is dedicated to monitoring the program execution in a totally non-interfering manner. Phase II simulates the environment under which the program was executed during phase I and provides the user with a **behaviorally equivalent** execution of the program (though not necessarily an identical execution). By behaviorally equivalent, we mean that all outputs of the reconstructed program are identical (in value, order of occurence and simulated time of occurence) to those of the original program.

The execution of a parallel program is viewed, at each processor site, as a sequence of macro-level instructions that are atomically executed. To reproduce the behavior of a parallel program across successive executions, it is not necessary to reproduce the execution of the same sequence of instructions. It is only necessary to ensure that only a certain subsequence of the macro instructions be executed in an identical order (with

respect to each other) during each execution. We refer to these instructions as *relevant actions* (relevant as far as behavior is concerned). We have proven that *reproducing an identical sequence of these relevant actions across successive executions is sufficient to ensure identical program behavior across these executions.* For example, all macro instructions operating on objects shared across processes are relevant since the sequence of operations (read,write) on these shared objects directly determines their values, and a different sequence could result in different values. We, therefore, detect and store information about the execution of instructions executing on shared objects for use in reproducing the behavior of the program. There are several other instructions, typical of real-time software, that are relevant. We shall list them and provide a complete theoretical framework for relevancy at the workshop.

As a direct result of this, we provide support only for identifying and gathering information about relevant actions during phase I. The relevant action sequence is collected at each processor site, in a structure called the event table. The event tables generated at the various sites can then be merged and used during phase II to perform the debugging either at one site or in a distributed manner. The principal issue of Phase I, therefore, is concerned with the hardware and software support needed to collect information about these actions *non-intrusively*. As Plattner [Plat84], we monitor the bus with a separate cpu and associated hardware. Unfortunately, it is impossible to gather even such elementary information as when a particular instruction is being executed on conventional cpu chips because of architectural features such as instruction pipelining and on-chip cache memories. It is necessary, therefore, that the bus be provided with additional bus lines to convey the necessary information about bus activity so that the monitoring hardware can know what to trap. Further details about the exact nature of the lines will be provided during the workshop.

Phase II is dedicated to ensuring that the execution of the program is behaviorally identical to that in phase I. We advocate a scheme similar to that proposed by Tai et. al [Tai86], except that ours is more general and complete in that it considers external interrupts and the fact that the program may be distributed across more than one processor. The program is executed under the control of a monitor which schedules the processes of the program in a manner that ensures an identical sequence of relevant actions, and hence an identical behavior, to phase I. The monitor also ensures an identical environment (e.g, external interrupt activation and clock input values) to that in phase I. Further details will be provided at the workshop.

The two phases outlined above guarantee a total reproducibility of behavior across successive exections. The whole technique, therefore, sequentializes a distributed execution, thereby reducing the complexity of debugging in these domains to that of debugging a sequential program!!

Checkpointing

There is, unfortunately, a practical problem in implementing the above strategy, namely, that the memory to store the relevant event sequence is limited and will eventu-

ally be exhausted. What is needed, therefore, is a scheme to checkpoint the program from time to time, and maintain the relevant event sequence from the last checkpoint. When an error is detected during phase I, the system is restored to the state reflected by the last checkpoint and a behaviorally equivalent execution simulated from then on, under user control. Of course, memory restrictions would still limit our ability to detect errors to only after the last checkpoint. Note that checkpointing must be done non-interferingly, in keeping with the overall goal of providing non-interference debugging.

To accomplish non-interferring checkpointing, we have developed a novel 2-level memory scheme to *checkpoint* programs *non-intrusively*. The memory architecture is coupled with the new bus architecture (mentioned above) to provide a way of obtaining a consistent state of the program when desired, **without perturbing** the target program!! We believe that such a memory architecture would help detect long term errors, a topic that has been completely ignored in the literature. More details of the architecture will be provided at the workshop.

Novel Contributions

The techniques described in this paper will make several novel contributions to the field, including:

- First two-phase non-interfering strategy of debugging with *identical* replay of execution.
- First strategy to support long term error detection.
- Identification of SUFFICIENT criteria for reproduciblity of parallel program execution.

Bibliography

[Bate83] Bates, P.C., Wileden J.C **High-level Debugging of Distributed systems : The behavioral abstraction approach** *CH2149-3/85 IEEE, pp498-506.*

[LeBl85] LeBlanc Richard J., Robbins, Arnold D., **Event-Driven Monitoring of distributed programs** *CH2149-3/85 IEEE, pp515-522*

[LeBl87] LeBlanc, T.J, Mellor-Crummey, John M., **Debugging Parallel Programs with Instant Replay** *IEEE Trans. on Computers, Vol. c-36, No. 4, April 87*

[Plat84] Plattner Bernhard, **Real-Time Execution Monitoring.** *IEEE TSE, Vol. SE-10, NO. 6, pp756-764, Nov. 1984*

[Tai86] Tai, Kuo-Chung, Obaid Evelyn E., **Reproducible testing of Ada tasking programs** *CH2281-4/86 IEEE, pp69-79.*

[Wits83] Witschorik, C. A. **The Real-Time Debugging Monitor for the Bell System 1A Processor.** *Soft. Prac. and Exp., Vol. 13, pp727-743, 1983.*

A Message-Based Approach to Distributed Database Prototyping

Sang H. Son

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

1. Introduction

As computers are becoming essential part of real-time systems, *real-time computing* is emerging as an important discipline in computer science and engineering. The growing importance of real-time computing in a large number of applications, such as aerospace and defense systems, industrial automation, and nuclear reactor control, has resulted in an increased research effort in this area. Since any kind of computing needs to access data, methods for designing and implementing database systems that satisfy the requirement of timing constraints in collecting, updating, and retrieving data play an important role in the success of real-time systems. Further evidence of its importance is the recent growth of research in this field and the announcements by some vendors of database products that include features achieving high reliability and performance [ACM88].

Performance evaluation of database systems has been studied by several researchers using mostly simulation and analytic models. However, few have addressed distributed environments, and the field of distributed database evaluation is currently in a state of disarray. Performance results are inconclusive and sometimes even contradictory [Wolf87]. We feel that an important reason for this situation is that many interrelated factors affecting performance (concurrency control, buffering schemes, data distribution, etc.) have been studied as a whole, without completely understanding the overhead imposed by each. An evaluation based on a combination of performance characterization and modeling is necessary in order to understand the impact of database control algorithms on the performance of distributed real-time systems.

Traditionally, database systems research has been concentrated in relatively specific areas such as file access methodologies, database language design, concurrency control and recovery, and query optimization. Surprisingly seldom, these separate components have been integrated in a single complete database system in a university environment. The reason is very simple: it is just too time consuming. By the time a complete system has been implemented, and is ready for performance evaluation, many of its component methodologies may have been superceded.

A prototyping technique can be applied effectively to the evaluation of control mechanisms for distributed database systems. A *database prototyping tool* is a software package that supports the investigation of the properties of a database control techniques in an environment other than that of the target database system. The advantages of an environment that provides prototyping tools are obvious. First, it is cost effective. If experiments for a twenty-node distributed database system can be executed in a software environment, it is not necessary to purchase a twenty-node distributed system, reducing the cost of evaluating design alternatives. Second, design alternatives can be evaluated in a uniform environment with the same system parameters, making a fair comparison. Finally, as technology changes, the environment need only be updated to provide researchers with the ability to perform new experiments.

A prototyping environment can reduce the time of evaluating new technologies and design alternatives. From our past experience, we assume that a relatively small portion of a typical database system's code is affected by changes in specific control mechanisms, while the majority of code deals with intrinsic problems, such as file management. Thus, by properly isolating technology-dependent portions of a

database system using modular programming techniques, we can implement and evaluate design alternatives very rapidly. Although there exist tools for system development and analysis, few prototyping tools exist for distributed database experimentation. Especially if the system designer must deal with message-passing protocols and timing constraints, it is essential to have an appropriate prototyping environment for success in the design and analysis tasks.

Another important use of a prototyping environment is to analyze the reliability of database control mechanisms and techniques. Since distributed database systems must work correctly even with subsystem failures, their behavior in degraded circumstances needs to be well understood. Although new approaches for synchronization, checkpointing, and database recovery have been developed recently [Son86, Son87, Son87b, Son88], experimentation to verify their properties and to evaluate their performance has not been performed successfully due to the lack of appropriate test tools.

2. Approach

When prototyping distributed database systems, there are two possible approaches: sequential programming and distributed programming based on message-passing. Message-based simulations, in which events are message-communications, do not provide additional expressive power over standard simulation languages; message-passing can be simulated in many discrete-event simulation languages including SIMSCRIPT and GPSS. However, a message-based simulation can be used as an effective tool for developing a distributed system because the simulation "looks" like a distributed program, while a simulation program written in a traditional simulation language is inherently a sequential program. Furthermore, if the simulation program is developed in a systematic way such that the principles of modularity and information hiding are observed, most of the simulation code can be used in the actual system, resulting in a reduced cost for system development and evaluation.

For a message-based simulation, the process view of simulation has been adopted. A distributed system being simulated consists of a number of *processes* which interact with others at discrete instants of time. Processes are the basic building blocks of a simulation program. A process is an independent, dynamic entity which manipulate *resources* to achieve its objectives. A *resource* is a passive object and may be represented by a simple variable or a complex data structure. A simulation program models the dynamic behavior of processes, resources, and their interactions as they evolve in time. Each physical operation of the system is simulated by a process, and the process interactions are called *events*.

A user can specify the number of processors, the number and locations of processes, the number and locations of resources, and the interaction between processes. We use the client/server paradigm for process interaction in our prototyping environment. The system consists of a set of clients and servers, which are processes that cooperate for the purpose of transaction processing. Each server provides a service to the clients of the system, where a client can request a service by sending a request message (a message of type *request)* to the corresponding server. The computation structure of the system to be modeled can be characterized by the way how clients and servers are mapped to processes. For example, a server might consists of a fixed number of processes, each of which may execute requests of every transaction, or it might consists of varying number of processes, each of which executes on behalf of exactly one transaction.

Internal actions of a process, i.e., actions that do not involve interactions with other processes in the system, are modeled either by the passage of simulation time or by the execution of sequential statements within the process. We use a simulator clock to represent the passage of time in a simulation. The simulator clock advances in discrete steps, where each step simulates the passage of time between two events in the system.

In a physical system, each process makes independent progress in time, and many processes execute in parallel. In its simulation, the multiple processes of a physical system must be executed simultaneously on one processor. This simultaneity is achieved by interleaving the execution of different processes and executing them in a quasi-parallel fashion. A scheduling primitive, **PauseProcess,** is provided to

guarantee quasi-parallel processing and finite progress of all active processes.

In message-based simulations, the communication of a message may take an arbitrary amount of simulation time. For instance, in our transaction execution example, we may assume that the time taken from sending a message until its reception by the destination process is insignificant; thus, in the simulation, the transmission time for the message that models this event can be zero. However, nonzero transmission delays exist in distributed systems, and they can be modeled by causing the process sending (or receiving) a message to wait for a certain time corresponding to the message transmission time before (or after) sending (or receiving) the message.

To develop an effective prototyping environment for distributed database systems, appropriate operating system support is mandatory. Database control mechanisms need to be integrated with the operating system, because the correct functioning of control algorithms depends on the services of the underlying operating system; therefore, an integrated design reduces the significant overhead of a layered approach during execution. There are several projects investigating methodologies for this integration of database functions with operating systems [Hask87]. Although an integrated approach is desirable, the system needs to support flexibility which may not be possible in an integrated approach. In this regard, the concept of developing a library of modules with different performance/reliability characteristics for an operating system as well as database control functions seems promising. Our prototyping environment follows this approach [Cook87]. It efficiently supports a spectrum of distributed database functions at the operating system level, and facilitates the construction of multiple "views" with different characteristics. For experimentation, system functionality can be adjusted according to application-dependent requirements without much overhead for new system setup.

3. Current Status and Plan

The implementation of the primitives required for message-based simulations, including constructs to create and terminate processes, to send and receive messages between processes, and to block a process for messages, data objects, or simulation time to elapse, is a critical component of the prototyping environment. In our prototyping environment, these primitives are being developed efficiently as part of concurrent programming kernel developed for the StarLite project. The module libraries for the distributed database prototyping environment are being designed and implemented in Modula-2 on Sun workstations. While Modula-2 has been chosen as the implementation language for our experiments, there are no Modula-2 dependencies in the program library. It is intended that the library be easily portable to other languages.

The current preliminary prototyping environment can provide multitransaction execution facility, including two-phase locking and timestamp ordering as underlying synchronization mechanisms. Replicated data object management and recovery logic has not been completed yet. The implementation plan of the prototyping environment is that we first focus on the development of basic modules and primitives. Then, we will prototype advanced techniques such as multiversion management using semantic information and time-driven synchronization mechanisms. One of the problems we are interested to investigate is the one referred to as *priority inversion.* Priority inversion is said to occur when a higher priority process is forced to wait for the execution of a lower priority process. When the transactions of two processes attempt to access the same data object, the access must be serialized to maintain consistency. If the transaction of the higher priority process gains access first, then the proper priority order is maintained; however, if the transaction of the lower priority gains access first and then the higher priority transaction requests access to the data object, this higher priority process will be blocked until the lower priority transaction completes its access to the data object. In order to maintain a high degree of schedulability, priority inversion must be minimized. An approach to this problem, based on the notion of *priority inheritance* has been developed at the Carnegie-Mellon University [Sha87]. The basic idea of priority inheritance is that when a transaction T of a process blocks higher priority processes, it executes at the highest priority of all the transactions blocked by T. The protocol using this idea of priority inheritance, called the *priority ceiling protocol,* not only minimizes the blocking time of a higher priority process to at most one

transaction execution but also prevents the formation of deadlocks [Sha88]. In addition to theoretical analysis of this protocol, an experimental evaluation is necessary to verify the properties of this protocol. We plan to use the prototyping environment being developed at Virginia for the evaluation of time-driven synchronization protocols including the priority ceiling protocol. If successful, the prototyping environment will show the possibility of sharing software and experimental results by researchers at different institutions.

References

[ACM88] ACM SIGMOD Record, Special Issue on Real-Time Database Systems, Vol. 17, No. 1, March 1988.

[Cook87] Cook, R. and S. H. Son, "The StarLite Project," *Fourth IEEE Workshop on Real-Time Operating Systems,* Cambridge, Massachusetts, July 1987, 139-141.

[Hask87] Haskin, R. et al., "Recovery Management in QuickSilver," *11th ACM Symposium on Operating Systems Principles,* Austin, Texas, Nov. 1987, 107-108.

[Sha87] Sha, L., R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocol: An Approach to Real-Time Synchronization," *Technical Report,* Computer Science Dept., Carnegie-Mellon University, 1987.

[Sha88] Sha, L., R. Rajkumar, and J. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record,* Special Issue on Real-Time Database Systems, Vol. 17, No. 1, March 1988.

[Son86] Son, S. H. and A. Agrawala, "An Algorithm for Database Reconstruction in Distributed Environments," *6th International Conference on Distributed Computing Systems,* Cambridge, Massachusetts, May 1986, 532-539.

[Son87] Son, S. H., "Synchronization of Replicated Data in Distributed Systems," *Information Systems 12,* 2, June 1987, 191-202.

[Son87b] Son, S. H., "Using Replication for High Performance Database Support in Distributed Real-Time Systems," *8th IEEE Real-Time Systems Symposium,* San Jose, California, Dec. 1987, 79-86.

[Son88] Son, S. H., "An Adaptive Checkpointing Scheme for Distributed Databases with Mixed Types of Transactions," *Fourth International Conference on Data Engineering,* Los Angeles, Feb. 1988, 528-535.

[Wolf87] Wolfson, O., "The Overhead of Locking (and Commit) Protocols in Distributed Databases," *ACM Trans. Database Systems 12,* 3, Sept. 1987, 453-471.

Butterfly™ HOSE: Graphical Programming for Parallel Systems

Hugh Sparks
MTS Systems Corporation
Eden Prairie, Minnesota

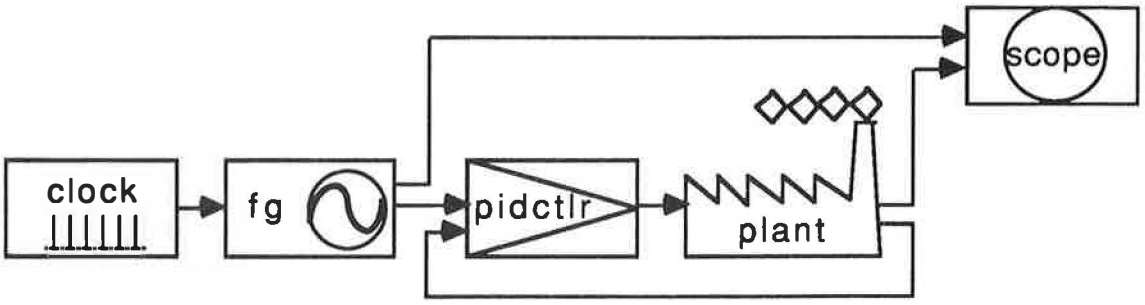
Bob Chatham
BBN Advanced Computers
Cambridge, Massachusetts

Abstract

HOSE (Hierarchical, Object-Structured Environment) is a graphical programming tool for designing and implementing parallel systems. Originally developed at MTS Systems Corporation, HOSE has been applied to a variety of complex real-time control problems. BBN Advanced Computers, in a cooperative development effort with MTS, is working to make HOSE more widely available using its Butterfly™ parallel processor as a hardware platform. This paper describes the current capabilities of HOSE, the Butterfly hardware, and plans for the next generation of HOSE on the Butterfly.

Introduction to HOSE

HOSE is a general-purpose tool for designing, simulating, and implementing large-scale parallel systems. HOSE programs are analogous to electrical circuit diagrams. In electronics, concurrent signals flow along wires between various components that modify the signals or generate new signals. Similarly, the connections of a HOSE diagram transmit data between computational objects called *primaries*. On a multiprocessor like the Butterfly, execution of these primaries can occur in parallel. A representative, high-level HOSE program is depicted below:



A HOSE diagram is both the design and implementation of a parallel system. It is not necessary for a system designer to translate a diagram into FORTRAN or some other procedural language in order to execute it. Special adaptations support real-time sensor data aquisition and control systems. Industry-standard hardware is used for sensor and actuator interface.

Hierarchical Design

HOSE diagrams are hierarchical. For example, the internal representation of the "plant" primary in the above diagram consists of other primaries. This feature of the language allows systems to be defined starting at the highest levels of abstraction. A design is then refined by drawing the internal structure of each component using more detailed data flow diagrams. A large library of primaries already exists, so users can create diagrams simply by selecting primaries from the library and connecting them. When an appropriate primary is not available, users can define their own with a Pascal-like language.

Object-Oriented Programming

When two or more instances of the same primary are used in a HOSE diagram, they share the same implementation code; however, each instance of the primary has a distinct internal state. For example, a HOSE diagram that controls a single robot arm may become one component in a larger HOSE diagram that operates multiple cooperating robot arms. The single-arm controller is used as a "class description" of which any number of separate instances can be defined.

HOSE primaries communicate by passing data and control information along the paths of the diagram. The behavior of primaries with respect to their inputs is defined in terms of procedures called *methods*. The arrival of data at a primary may cause one or more methods to be invoked. Methods propagate data flow by producing outputs which are sent "downstream" to other primaries in the diagram. Through its use of this data/message-passing scheme, HOSE retains the flavor of object-oriented systems such as the Smalltalk programming language.

HOSE Applications

HOSE diagrams are used primarily to design, simulate, and implement real-time control systems.To date, HOSE has been applied to a number of control problems at MTS. LARS (Laser Articulated Robotic System) consists of a 15 kilowatt CO_2 laser integrated with a robotic manipulator and vision system that can do metal cladding, cutting, or welding to a tolerance of +/- 0.005 inches at speed of up to 200 inches per minute. IRIS (Intelligent Robotic Inspection System) is a non-contact coordinate measuring system used for high-speed digitizing of large and complex parts surfaces.

HOSE Architecture

HOSE consists of two major subsystems: a front-end consisting of the diagram-editing interface, and a back-end, or kernel, which coordinates the flow of data through the diagram. The kernel runs on an MTS-designed multiprocessor (described below), and the interface runs on a Digital VAX™. Digital VT-240™ terminals are used to edit diagrams. A communcations link with a simple command protocol connects the two subsystems. It is possible to simulate the multiprocessor by running both interface and kernel on the VAX. In this case, the communications link is replaced by interprocess communication via mailboxes, or is done away with altogether, and calls are made directly into the kernel.

The current HOSE kernel engine is a bus-based multiprocessor consisting of as many as five Motorola 68020 processors sharing a VME bus with a large block of global memory and sensor/actuator interfaces. Through the use of bus interconnects, larger configurations of as many as 26 processors have been assembled. However, the technical complexity and performance constraints of larger bus-based systems have lead MTS to seek an alternative hardware solution through the use of BBN Advanced Computers' Butterfly parallel processor.

The Butterfly Parallel Processor

The Butterfly is a tightly-coupled, multiple-instruction multiple-data (MIMD) machine which can be incrementally configured with between 3 and 256 processing nodes. Each node consists of a Motorola 68020 processor with 68881 floating-point coprocessor, 4 megabytes of RAM, and custom switching hardware which allows all processors to transparently access the memories of all other processors. In its largest configuration, the Butterfly delivers an aggregate 600 MIPS.

The Butterfly Switch

The Butterfly switch is a collection of 4-input, 4-output switching elements configured as a serial decision network. Using techniques borrowed from packet-switching technology, it implements fast, reliable, and economical interprocessor communication. The topology of the switch resembles that of a Fast Fourier Transform Butterfly, hence the name. Non-local memory references through the switch take approximately 4 microseconds. This fast remote access time promotes the use of the Butterfly as a machine with a large shared memory, simplifying the job of parallel programming.

Butterfly I/O

The Butterfly supports both VMEbus™ and Multibus™ peripherals. A Multibus adapter card fits into the Butterfly computer rack and connects to the I/O bus of each processor

node. The VME bus connects directly to the Butterfly switch. Multibus I/O devices communicate with a processor node via a synchronous 2Mbyte/s bus called the Butterfly I/O link (BIOLINK). Through the BIOLINK, each Butterfly processor can directly access memory on its attached Multibus, as well as read or write the control and data registers of peripheral devices on the Multibus. A Multibus device can directly read or write the 4Mbytes of RAM on the processor node to which the adapter is attached.

The Butterfly VME interface (BVME) accomodates I/O devices whose bandwidth requirements exceed the normal capacity of a processor node. A single BVME adapter connected to two switch ports provides 5.5 Mbytes/s of typical data throughput.

The Butterfly HOSE Project

The Butterfly HOSE project is just getting underway as of February, 1988 but several goals have been established:

• Port HOSE's current functionality to the Butterfly

In order to quickly establish that HOSE on the Butterfly is a viable concept, the first stage of the project involves simply moving the existing functionality of the MTS HOSE kernel to the Butterfly. At this stage of the project, no modifications will be made to the user interface other than to port it to a Sun workstation. At the conclusion of this phase, a version of Butterfly HOSE will exist that is source compatible with the current MTS version of HOSE.

• Develop performance monitoring tools

To facilitate the task of building parallel systems, we are going to develop tools to help users debug and tune HOSE programs. Of particular interest are tools for load balancing and performance monitoring. In terms of HOSE primaries, load balancing means the assignment of methods to particular processors. Although static allocation is currently used, the code for all methods resides on all processors. Since there is a uniform cost associated with sending data to any processor node, it is possible that dynamic allocation of HOSE methods to processors can take place.

• Modify the user interface to run under X Windows

Porting the interface to X makes it possible to run the HOSE diagram editor on a wide range of hardware. The interface will be modified to incorporate MacDraw®-like capabilities, and support for color will be added. Ideas involving the adaptation of HOSE diagrams for animated simulation have been discussed. It is already possible to hook up a probe to a "wire" in a HOSE diagram and connect the wire to a software "oscilloscope" to view data traveling between diagram components. This work would involve allowing users to create new kinds of animated objects (meters, dials etc.) that could be added to HOSE diagrams to help visualize the state of a system.

- Support development of primaries in languages other than Pascal

Currently, primaries and their associated methods are written in a language derived from Pascal. It would be advantageous to allow the use of other languages for primary definition.

PANEL

REAL-TIME APPLICATIONS ISSUES

Chairman

Pat Watson
IBM Federal Systems Division

Panelists

Marc Donner
IBM Research

Michael Hawley
Massachusetts Institute of Technology

Lee Lucas
Navel Weapons Center

Thomas Ralya
IBM Federal Systems Division

Nicholas Baker
McDonnell Douglas Company

Arun Venkatraman
National Astronomy & Ionosphere Center

NOTES:

NOTES:

SESSION V

REAL-TIME APPLICATIONS

Chairman

Michael Hawley
Massachusetts Institute of Technology

Functional vs. Object-Oriented Development of Robot-Control Software (A Comparison of Two Robot-Control Programs)

Thomas E. Bihari
Adaptive Machine Technologies
1218 Kinnear Road
Columbus, Ohio 43212
(614) 486-7741

The Adaptive Suspension Vehicle (ASV) is a three-ton, six-legged walking robot designed to traverse rough terrain. The ASV was designed and built by Adaptive Machine Technologies, in cooperation with the Ohio State University, under a contract with the Defense Advanced Research Projects Agency (DARPA). The ASV is propelled by 18 hydraulic actuators, powered by an internal combustion engine. The ASV sensor suite includes leg position sensors, pressure sensors on the hydraulic actuators, an inertial sensor system on the body, and a laser range-finder which generates 3-D images of the terrain ahead of the vehicle. A complex control program processes sensor inputs and generates actuator commands, allowing the on-board operator to control the ASV via joystick inputs, in a "fly-by-wire" mode.

The High-Performance Manipulator (HPM), also sponsored by DARPA, is a six degree-of-freedom, hydraulically-actuated robot arm with a 2.5 meter reach, designed for high-speed operation (2 g end-effector acceleration), with the ability to manipulate payloads (over 200 kg) weighing a substantial fraction of the weight of the arm itself (approximately 400 kg), while maintaining good repeatability (approximately 1 mm). The HPM has been designed so that it may be easily mounted on a mobile platform.

The ASV software development effort, which was begun in 1983, was time-consuming, due to the size of the ASV control program (over 150,000 lines of Pascal source code), and the complex, time-dependent interactions among software and hardware components. Sensory and control algorithms were changed frequently, based on improved understanding of the characteristics of the underlying mechanical system. Furthermore, the mechanical hardware was also under development, and new missions (e.g., terrain obstacles to cross) were being tackled. These factors led to an "experimental" or "iterative" approach to software development, in which software components were continually

modified or replaced. As the ASV program progressed, it became difficult for programmers to fully comprehend the operation of the complete control program, and how modifications to certain software components affected other components.

In 1987, work began on the HPM control program. Based on experience with the ASV software, which used a traditional "functional" software development methodology, the decision was made to use an object-oriented development methodology for the HPM software.

The ASV and HPM control systems are similar in many respects. Both use embedded multiprocessors for computational power. The ASV control computer contains 14 Intel 8086 processors, and several special-purpose processors for vision-processing and inverse-dynamics calculations. The HPM control computer is an Intel 80386-based multiprocessor, essentially an upgraded version of the ASV control computer, with correspondingly updated special-purpose processors. Both control computers host versions of the GEM operating system, developed at the Ohio State University. Both incorporate sophisticated force-control algorithms which require high servo rates. The ASV program contains vision-processing and leg-interaction software which is not needed by the HPM. However, servo-level and motion-planning-level software is conceptually very similar.

Experience with the "functional" ASV control software and the "object-oriented" HPM control software has been instructive. The similarities in the underlying control algorithms allow the implementations to be compared side-by-side. Furthermore, the same programming team has been responsible for both projects, providing a good deal of feedback during the design of the HPM software. While there was some initial confusion, as engineers attempted to "think functionally" while designing objects, the transition to object-oriented programming has gone quite smoothly. The resulting HPM control program consists of layers of "virtual arms", with each successive arm layer hiding a level of control. The human controller interacts with the top-level arm via a relatively "natural", high-level interface.

In addition to easier prototyping and improved software reuse, object-oriented development has led to a better conceptualization of the HPM control task. AMT expects to use an object-oriented methodology for the development of the Advanced Ground Mobility System (AGMS), a DARPA-sponsored, second-generation walking robot, which is currently in the conceptual-design phase at OSU and AMT.

DESIGN AND IMPLEMENTATION OF A REAL-TIME MULTIVARIABLE ADAPTIVE CONTROLLER

Gianfranco CICCARELLA
Scuola Superiore G. Reiss Romoli, Via di Coppito 67100 L'Aquila and Department of Electrical Engineering, University of L'Aquila, ITALY

ABSTRACT.

This paper discusses the main design and implementation problems of a multivariable adaptive controller for continuous, linear, time invariant, dynamic systems. The controller (Fig. 1) is based on a reference model and has been developed taking into account both the control problems (such as the influence of the discrete control on the behavior of continuous systems), and the need of executing the algorithm with a high degree of parallelism in order to match the real time constraints even in complex applications. The paper discusses the real time implementation of the control algorithm on embedded multiple processor systems and shows that some design steps can be simplified if a concurrent programming language (Occam) and a VLSI processor (Transputer), designed by a language first type approach, are used.

Keywords: Real-Time Software, Adaptive Control Algorithms, Microcomputer Networks, Concurrent Programming.

1.INTRODUCTION

Real time industrial control systems are generally modeled as a set of processes with constraints on their execution time [1,2]; in fact the reaction to an incoming event must be guaranteed before a given maximum delay. A process corresponds to the execution of a program that performs a given function (data acquisition, control inputs synthesis, etc...) and that is usually executed periodically.
A synchronization graph G [1] can be used to represent the processes structure of a cyclic real-time system. A node of G (Fig. 1. [2]) can represent either a program Pi or a timing constraint Ti; a node labeled with a program name indicates the program execution and therefore represents a process, while a node labeled with a timing constraint represents the timing constraint for the processes which converge in this node.
In the example of Fig. 2. the programs P1-Pn are concurrent and must complete by time T2: they can then be executed on different processors to satisfy timing constraints.
The concurrence, in control applications, is often needed mainly to allocate several processes to the same system function in order to get it done faster. But a parallel implementation of the control software requires the development of suitable algorithms and hardware architectures.
The main design problems of multiple processor systems for control applications can then be summarized as follows :

- definition of suitable control algorithms, that allow a parallel implementation (to satisfy real-time constraints);
- choice of methods to proceed from the algorithm to the hardware-software design capable of being implemented on multiple processors systems.

In this paper we refer to the last point and analyze the implementation of an adaptive controller for continuous, linear, time-invariant, dynamic systems [2,3,6] using standard microprocessors and Transputers and the Occam language [4,5]. This language and this VLSI component seem to offer an effective solution to the design problems, as they integrate the programming language, the operating system and the processor hardware architecture.

Section 2. briefly discusses the concurrence of the adaptive control algorithm; section 3. analyzes the problems related to the implementation of real time control systems and shows that some design steps can be simplified if a concurrent programming language (Occam in our case) and a VLSI processor (Transputer), designed by a language first type approach, are used.

2. THE MULTIVARIABLE ADAPTIVE CONTROL ALGORITHM.

This section presents the synchronization graph of the adaptive controller to show its high degree of parallelism. We refer to [2,3] for a complete description of the adaptive control scheme, the design hypotheses, the control inputs synthesis and the parameters identification algorithm.
The control inputs vector, for a zero order (sample and hold) interpolator, is given by:

$$u(k-1) = D^{+}[x_M(k) - qx_M(k-1)] + D^{+}[qI - E]x(k-1) \qquad (2.1)$$

where :

- x_M and x are the model and system state variables;
- q is a scalar constant, $|q| < 1$;
- $D^{+} = [D^{T} D]^{-1} D^{T}$ is the left pseudo-inverse of D;
- E and D are the estimates of the submatrices of the plant augmented state transition matrix and characterize the plant to be controlled.

Fig. 2 shows a synchronization graph for the adaptive controller:

- the elements of the E and D matrices can be estimated by rows (processes P_1 ,P_2 , ... P_N), and also the estimation algorithm can be implemented using several concurrent processes;
- the control inputs vector u(k) is computed by PU1 and PU2, that calculate D^{+} and equation (2.1);
- the reference model state variables $x_M(k)$ are computed by the PM process.

A more efficient synchronization graph is shown in Fig. 3.
In this case processes P1-Pn have all the frame time for the execution, but it is necessary to use the estimates of the previous sampling period to synthesize the control inputs vector.
Simulations have shown that this delay modifies only the first values of the u(k) vector.

3. ADAPTIVE CONTROLLER IMPLEMENTATION.

In this section we deal with the hardware and software design of multiple microprocessor systems for real time process control applications. We refer to our experience [2,6,8] in the implementation of real time systems using standard microprocessors and Occam and Transputer based systems.

The following steps summarize the main design problems of real time computing systems for control applications:

a) choice of the control scheme and of the control algorithms;

b) definition of time constraints and reliability requirements, which influence the software and hardware architecture;

c) definition of an application software model using synchronization graphs;

d) choice of the computing system hardware and software architecture, taking into account problems related to the operating system and fault tolerance;

e)validation of the constraints on response time and on recovery time. If the validation is satisfied go to point i), else f);

f) choice of a control software model that, using concurrence, decreases the time complexity;

g) modification of the hardware architecture to increase parallelism;

h) go to point e);

i) hardware and software implementation of the control system.

Some steps of the top down design can be simplified using Occam and the Transputer. The hardware and software architectures are in fact defined by the Occam description that is mapped on a Transputer network for the implementation.
At the lowest level Occam is a language to program arrays of Transputers [5], but it can also be used as a design tool (system description language) capable of describing all the stages of a design: from an abstract high level description of the software and of the hardware architecture, down to a program. The modification of the software architecture, after the validation of the constraints on response time, is obtained by a mapping on a different network with a bigger number of transputers. In fact an Occam program can be executed unchanged by a single computer or by a network of computers: an Occam process can communicate with other processes only by means of channels, which are implemented by values in memory (for processes allocated on a single computer) and by physical links (for processes allocated on different computers).

With reference to the implementation, i.e. to the mapping of the Occam description on a network of Transputers, the main remarks are the following:

- choices made for the software architecture give constraints on the network topology; so it is necessary to define the structure of the Occam programs in such a way that they can be efficiently mapped on a network of transputers;
- it is necessary to distribute the computational load so that all Transputers have high active time and must not wait for input data from other processors. The ratio of active time to total time for the complete system is a measure of its efficiency;
- communicating processes should be allocated on Transputers that are directly connected in order to decrease the communication overhead.

The implementation by using single board computers based on conventional microprocessors [2] required:

- the interfacing to an operating system kernel;
- more complex prototyping and debugging;
- a remarkable complexity in passing from the single processor system to a multiprocessor one;
- a bigger system developing time.

4. CONCLUSIONS.

This paper has the aims to emphasize the needs, in real-time control applications, of an integrated hardware and software development environment and of suitable algorithms that allow parallel implementation. We have discussed the design and implementation problems on multiple processor architectures of a real-time model reference multivariable adaptive controller.

The main advantages related to the use of a concurrent language (Occam) and of a VLSI processor designed by a language first approach (Transputer) are the following:

- the use of a single language for the architecture definition and for the processor programming;
- the integration among programming language, operating system and processor hardware architecture.

Experience acquired using Occam and Transputers has also shown that it is necessary:

- to define and develop software modules for testing response time in communication primitives, for managing scheduling policies and for implementing interprocess communication;
- to define and develop hardware and software fault tolerance techniques for transputer networks;
- to define methods that allow to choice the structure of the Occam program that can be efficiently mapped on a network of Transputers. We are now studying general criteria to evaluate the advantages and the disadvantages of different parallel implementations taking into account the remarks made in section 3 about efficiency, load balancing and communication overhead.

REFERENCES

[1] T. Anderson, J.C. Knight, A Framework for Software Fault Tolerance in Real Time Systems. IEEE Trans. on Soft. Eng, May 1983.

[2] G.Ciccarella, G.Del Maestro, D.Di Domenico, Microcomputer Implementation of Adaptive Controllers for Continuous Systems. Int. Journal of Microcomputer Applications, no.3, 1985.

[3] G.Ciccarella, G.Del Maestro, D.Di Domenico, An algorithm and a Multimicrocomputer Architecture for Multivariable Adaptive Control. ISMM International Symposium on Mini and Microcomputers and their Applications, Bari, Italy, June 5-8, 1984.

[4] Inmos, OCCAM Programming Manual. Prentice Hall, 1984.

[5] Inmos, Transputer Reference Manual, September 1985.

[6] C. Cecati, G.Ciccarella, Occam and Transputer Based Systems for Process Control Applications. RAI/IPAR '86, AFCET-IASTED Conference on Robotics, Artificial Intelligence, Identification and Pattern Recognition, june 1986, Toulouse France.

[7] D.Landau, Adaptive Control. The Model Reference Approach. Marcel Dekker Inc. , 1979.

[8] C. Cecati, G. Ciccarella, P. Di Felice, Architectural Issues in Multiple Microprocessor Systems for Process Control. VI International Conference on Control Systems and Computer Science, Bucharest, May 22-25 1985.

[9] G.Ciccarella, P.Marietti, C.Padovani, Low Cost Access Units for Local Networks. IEEE Mediterranean Electrotechnical Conference, Athens, May 24-26, 1983.

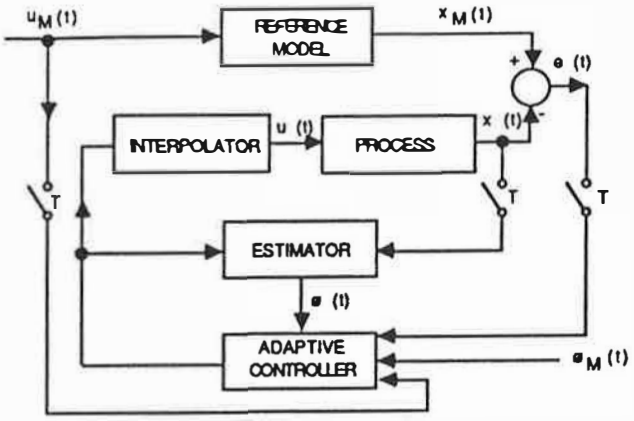


Fig. 1 - Adaptive control scheme

$$\dot{x}_M = A_M x_M(t) + B_M u_M(t) \quad ; \quad x_M(0) = x_{M0} \quad \text{(model)}$$

$$\dot{x}(t) = Ax(t) + Bu(t) \quad ; \quad x(0) = x_0 \quad \text{(system)}$$

$$u(K-1) = D^{+}[x_M(K) - qx_M(K-1)] + D^{+}[qI - E]\,x(K-1)$$

Estimation algorithm.

$$\beta_i^T(L) = [\,\varnothing_{i1} \ldots \varnothing_{in}\ a_{i1} \ldots a_{iq}\,] \quad \text{(parameters)}$$

$\varnothing_{ij}$, a_{ij} : coefficients of the i-th row of the E and D matrices

$$w^T(L) = [x_1(L) \ldots x_n(L)\ u_1(L) \ldots u_q(L)] \quad \text{(measurements)}$$

$$\beta_i(L+1) = \beta_i(L) + P_i(L+1)w(L+1)[x_i(L) - w^T(L+1)\,\beta_i(L)]$$

$$P_i(L+1) = c^{-1}[P_i(L) - r_i(L)\,P_i(L)w(L+1)w^T(L+1)\,P_i(L)]$$

$$r_i(L) = [c + w^T(L+1)\,P_i(L)w(L+1)]^{-1}$$

$r_i \in R^1$, the c coefficient is $0 < c \leq 1$

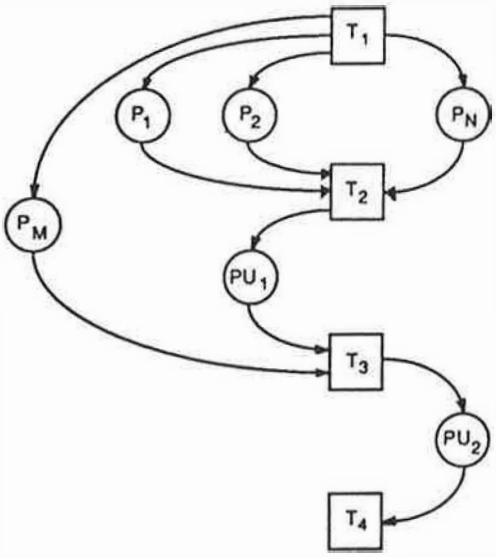


Fig. 2 - Synchronization graph of the adaptive control algorithm

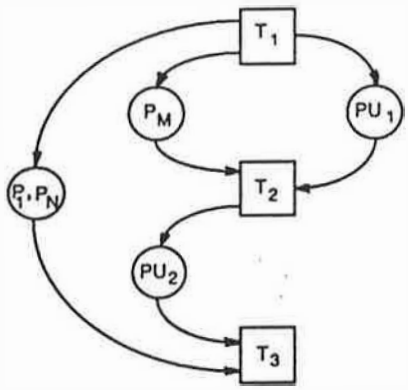


Fig. 3 - A different form for the synchronization graph of Fig. 2

Real-Time Operating System Architecture Worksteps and Related Subjects

Tom Ralya

International Business Machines
Federal Systems Division
9500 Godwin Drive
Manassas, Virginia 22110

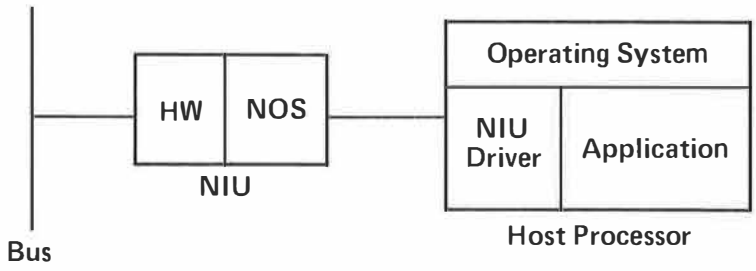
1.0 Introduction

Since 1978, IBM has developed several Local Area Network (LAN)-type data buses for real-time systems. In each case, the LAN was designed for high volume on the bus and through each node. Bus bandwidths have been in the range of 4 to 8 megabytes per second with individual node bandwidths in the range of 2 to 5 megabytes per second. All buses have been designed with peer-to-peer message-passing service with hard real-time deadlines. In the development of the microprocessor control software for the network, a serious shortfall in performance was experienced. The software, which is called Network Operating System (NOS) in this discussion, underwent a major redefinition of its architecture designed to guarantee response times to hard deadline users. In addition to the guaranteed response times, the new architecture has shown itself to be approximately 60 times faster. This report covers the attributes of the new architecture and the levels of service that have been developed. This architecture encompasses a workstep concept, an approach to dynamic memory management and modeling, as well as hardware assistance.

2.0 Background

The current version of the IBM real-time LAN is called the Real-Time Communications Network (RTCN) and consists of multiple, interconnected, linear buses. Individual nodes on the LAN are either bridges, host computers with a LAN attachment hardware set or Network Interface Unit (NIU), or nonprocessor devices.

The NIU contains frame-control hardware, speed-matching buffers, and a microprocessor that provides network control, user command interpretation, and multiplexing. The control software in the NIU is called the NOS. The NIU is attached to a host computer memory bus as an intelligent Direct Memory Access (DMA) controller. The host computer contains an operating system with software called the NIU driver, which is similar to a device driver. The NIU Driver adapts the NIU software interface to the actual application user software interface. One node of a network is shown in the following block diagram.



The NIU and its software, the NOS, provide a general message-passing service to application software running in an attached device. The message passing is based on a concept that includes Input/Output (I/O) channel-type operation called connection service. The connection service is offered as three basic types of connections to application user software. The most complex connection is similar to the International Standards Organization (ISO) model, Level 7 application service, in which the user's view is a single stage of a large pipeline process. The user processes are tightly time-synchronized (within 1 millisecond) periodic processing tasks. During each period, the NIU fills input buffers and empties output buffers over a set of network connections, while each user processes input buffers into output buffers. This type of service, called simplex in RTCN, is dominated by the hard deadline represented by the end of the time cycle.

The second type of connection is similar to a Level 5 session service and is called half duplex. With a half-duplex connection, a user may start a transaction and serially send or receive messages from the user on the other end of the connection. The message lengths and direction of data flow must match on both the send and receive commands or the transaction is terminated by the network. The connection

is maintained by the network. In the current RTCN, this service is scheduled by a priority value from 0 to 255. If that value is derived from response-time requirements, RTCN will deliver guaranteed response times.

The third type of connection is similar to a Level 4 transport service and is called duplex. With a duplex connection, a user may concurrently send and receive buffers of data. The user has no visibility into the buffer boundaries at the other end of the connection. In the current RTCN, this service is scheduled by a priority value from 0 to 255. If that value is derived from response-time requirements, RTCN will deliver guaranteed response times.

In addition to connection service, the RTCN includes connectionless service. A user may send a short mail message, limited to 2,000 bytes, addressed to a single user name or a group name. When mail is addressed to a single user, the sending user may request an acknowledgement of delivery. Mail services are generally scheduled by a priority value.

The introduction referred to a serious shortfall in performance of the NOS software. This occurred when the simplex and half-duplex software had been developed and tested. The performance test showed that NOS produced only 7 percent of the specified throughput. The performance test of the new architecture showed that the NOS had been improved to 430 percent of specified throughput.

This description of NOS could apply to almost any software that provides a command/response service to other software entities. For this reason, the architectural characteristics that helped achieve the dramatic increase in throughput are valuable for development of other application software.

3.0 Workstep Concept

The single most dramatic change to the NOS software was achieved in the area of how to share the Central Processing Unit (CPU) between different tasks. The key to sharing the CPU is in the selection of a tasking model. In tasking-model selection, key areas must be considered such as the fact that the sharing of the CPU for any real-time system must be efficient (i.e., low overhead) and it must be predictable. The early RTCN efforts expended considerable resources in order to guarantee service response times for sharing the bus resource. The new architecture extended these principles to the sharing of the CPU and memory for the NOS software.

With the new architecture, tasks were defined under design constraints making them very different from the more classic definition of tasks that had been used previously. In order to emphasize the fact that these were very different types of tasks, they were not even called tasks; they were called worksteps. A workstep is a sequence of instructions which is executed, from entry to exit, without interruption. Input to a workstep is always a single data object. A workstep is not allowed to pause or wait in any way for some external event to occur before continuation of processing. As such, a workstep represents a single state change to the input data structure and possibly to other data structures. The input of a single data object is a classic definition of a data-driven architecture.

These worksteps have several very interesting characteristics. The worksteps are:

- Uninterruptible and thereby limit the number of different states that must be accommodated;
- Limited in execution time by their design to allow higher priority work to gain control of the CPU;
- Scheduled by priority which can be a function of required response time;
- Data driven with the priority tied to the data instead of the task; and,
- Dispatched very efficiently.

3.1 Uninterruptible

The benefits of execution in an uninterruptible state are best summed up in four characteristics. First, without interrupts, the entire set of instructions represents a single change of state. Second, since each workstep is a single change of state, the number of different states of the software is greatly reduced. Third, the fact that the individual worksteps complete their duties before they exit means that instructions are not spent on saving and restoring the task environment such as registers and instruction address. And fourth, the execution time of each workstep can be predicted and controlled.

3.1.1 Complete Change of State

A great deal of state-of-the-art real-time programs are written in a manner that allows the halting of one task in order to service an interrupt. Sometimes that interrupt is less important than the task that is halted, which means that "more important" work is delayed because of "less important" work. Sometimes the interrupt is more important than the task that was halted, which leads to the suspension of the interrupted task at any point in its sequence of instructions. When a task gets suspended, its state, when referenced by another task, is neither its final state nor its initial state; it is in some intermediate state. The number of intermediate states is determined by the number of places in its sequence of instructions where it might be suspended. For all practical purposes, the task has as many states as it has instructions. In contrast, a workstep has its initial state and its terminal state; no intermediate states exist.

The benefit of allowing interrupts is obvious: the truly higher priority interrupt is serviced immediately. But there are very real costs with this approach. In order to service the interrupt, the current processor state must be saved. Some processor hardware provides two sets of registers to reduce this cost for temporary interruptions; some do not. The real issue centers on what state the processor is in when the interrupt is serviced. Since the processor state is the collection of task states, the processor could be in almost any state when the interrupt is being serviced. Therefore, the software tasks and the interrupts themselves must be written to allow that co-resident tasks may be in any one of an infinite number of states and that other tasks may change the data being worked on as the work is being performed. The result is the use of access flags, synchronization points (i.e., rendezvous), and other mechanisms. Simply stated, the workstep architecture does not allow this condition to exist.

The workstep architecture has, as its core, a workstep dispatcher. The dispatcher will first service interrupts only long enough to determine which workstep must execute for each active interrupt. The dispatcher then chooses and dispatches the highest priority workstep. Interrupts are not allowed while the workstep is executing. Then, the workstep returns to the dispatcher and the cycle begins again. Using this approach, whenever a workstep or an interrupt handler is executed, other worksteps have previously reached a stable, consistent state. This one simplification to both the run-time environment and to the programmer's development job was completely underestimated by programmers who were familiar with the task-oriented architectures. The results of this simplification are significant.

In actuality, it is likely that the workstep dispatcher will service "less important" interrupts while "more important" worksteps wait to be dispatched. As long as the priority of an interrupt is not determined, it must be treated as a top priority job. However, rather than completely servicing the interrupt before dispatching another workstep, the dispatcher's job is to determine the priority at which the service must be performed. It will then enqueue the proper workstep at the proper priority to actually service the interrupt; interrupt handlers are worksteps too. This is the minimum that must be performed, and even this may be eliminated through the use of special hardware assist. (See Section 6.0 Hardware Assist.)

3.1.2 Finite Number of States

Since each workstep represents a single state change, the total number of states of the processor is something that can be understood and represented. An early version of RTCN NOS contained 160 worksteps using 196 priorities. This is higher than the normal number of tasks in a multitasking system. Without task control blocks, the overall processor job is reduced to one workstep-size state change at a time. This view of the processor job is the key to modeling processor performance, easier understanding of the processor's job by system engineers and hardware designers, isolating testing down to individual worksteps, and simplifying integration.

Modeling processor performance is discussed in a later section. The understanding of the processor's job is enhanced by the fact that the processor functions can be represented as a set of state changes. System engineers and hardware designers are used to working with state change diagrams. The RTCN project, at several points in the development, performed hardware/software trade-off studies. The change diagrams enable hardware engineers to understand how a function fits into the whole picture, and with this knowledge, they are more helpful.

The single state change nature of worksteps means that when a workstep begins its execution, all other worksteps are in their initial state. The data structures that must be referenced by the workstep are always in one of a finite number of states. Testing can be reduced to running the workstep with each of the legal states of input data. The interruptible tasks could accomplish the same thing through the use of critical sections of code or locking semaphores. However, these mechanisms may actually complicate the attempts to model the result.

Even though the NOS function never faced the integration of multiple programs within the same processor, it would seem that the testing simplification would also extend to this type of integration testing. The integration task of testing different possibilities would seem to be greatly simplified when each program being integrated is in a legal state before another program runs and shares no resource beyond the CPU during run time (while the CPU sharing is managed by guaranteed response times).

3.1.3 Eliminates Saving Task Environment

Since worksteps execute to completion, they leave no information in registers that must be saved and restored on their next execution. The microprocessors that have been used in RTCN are from the Motorola 68000 family and do not provide two sets of registers to reduce the interrupt-handling overhead. The original NOS design, if it could have run at specified performance levels, would have used 5 to 10

percent of the CPU to the save and restore states. The workstep architecture saves all of this overhead.

In fact, with the use of a generalized multitasking executive, the overhead cost of task switching was several hundred microseconds for each switch. This would have set the task switching overhead to 30 to 50 percent of the CPU. By contrast, the workstep architecture requires each workstep to be completely finished with the registers, before termination. Workstep dispatcher overhead is reduced to figuring out which workstep has highest priority and passing its data structure pointer to the workstep. The RTCN NOS workstep dispatcher uses less than one percent of the CPU when executing at specified performance levels.

3.1.4 Predictable Execution Time

The predictability of task and workstep execution time seems, on the surface, to be equivalent. However, the RTCN experience points out some subtleties that are easily overlooked.

Why is predicting execution times for code in a real-time system necessary in the first place? Generally, the first concern is that limited CPU resources and overloads cannot be allowed. Additionally, in a real-time system, events must happen within specified time constraints. The ability to produce code that meets the time constraints depends on the ability to predict execution times, unless one is willing to use an iterative approach to meeting response time requirements.

To predict the response time to an external stimulus, a task is broken down into sequentially executed segments of code with delays for subevents between segments. Each segment has a start point, an end point, and time to execute is predicted. But the code must execute in the presence of contention from other tasks. What happens to the execution time in this environment? The developing programmers generally find great difficulty responding to this question. The partial execution, suspension, and later, continued execution, are difficult to separate due to the difficulty in determining when a particular subevent has occurred. The difficulty is caused by the probability that the code creating the subevent is in some intermediate state. Relationships between different tasks are very difficult to describe and predict.

The result of the difficulty in producing execution time predictions is that either the predictions are done poorly or not at all. This is not suggesting that the programmer is somehow at fault. Programmers are reluctant to say, "This code takes X milliseconds," because saying this would be so heavily dependent upon events external to their code. Programmers are isolated from understanding the execution of the code and, therefore, must either reduce the feeling of responsibility or must feel responsible for something that is outside of their control. Neither result is desirable.

The workstep architecture separates the variable elements of predictability from the execution of any one workstep. The overall question of an external stimulus response time for the workstep architecture is broken down into the worksteps (roughly equivalent to the code segments previously mentioned), delays for subevents, and delays in scheduling due to other workstep execution. Subevent determination is made almost trivial because a specific workstep will cause the subevent. Programmers are held responsible only for the execution of their

worksteps. These execution times are all that is necessary for lead programmers and system engineers to model and predict the stimulus response times.

3.1.5 The Dilemma

Up to this point, this entire section has pointed out benefits of the uninterruptible nature of the workstep architecture without discussion of the obvious problem. If worksteps cannot be interrupted, then "less important" work can hold off "more important" work long enough for response-time requirements to be missed. This is, of course, totally unacceptable. But the key to the problem is in the words, "long enough." To determine how long is long enough, the architecture incorporates a rule that all worksteps be designed to limit their longest execution time. This time limitation is an absolutely basic part of the workstep architecture. The next section discusses the problems and solutions in the imposition of the time-limit rule.

3.2 Execution Time Limited

The primary rule in the workstep architecture is the execution time limitation. Each workstep must be designed so that its longest execution path does not exceed a specified time limitation. On RTCN NOS, the time limitation was 2 milliseconds, but other applications need to select their own time limitations.

The NOS design team considered implementing the time limitation through the use of a watchdog timer at execution time instead of a design limitation. However, to do so would eliminate the benefit of reducing the number of states discussed in the previous section.

Therefore, our choice was to place a design constraint on all programmers. During the NOS development, the single most difficult thing for the RTCN NOS programmers to obey was getting used to this rule. Even with a strong degree of attention, some took as long as three months to acclimate themselves to the execution time during the design phase of development. However, in the long run, their attention carried over into the coding phase and paid large dividends. Programmers took pride in reducing the execution time of their worksteps.

Trade-offs between time and storage were performed in a clear atmosphere. The model identified the time-critical worksteps and the effect of reducing execution times for those worksteps, thus allowing informed quantitative judgments to be employed as to where to spend extra time in optimization. Development of worksteps that were infrequently executed or that did not affect critical response times could be optimized for development schedule or for storage use. One of the most surprising experiences during the development of RTCN NOS was that intuition is not a good guide to selection of non-critical worksteps.

3.2.1 Not Hard to Obtain

A natural tendency for most programmers is to declare that the limitation will cause hardship and lower productivity rates. In fact, this reaction cannot be supported, at least for the RTCN effort. The NOS is basically an I/O-bound service processor. Due to that fact, each service request broke down into a sequence of processing, followed by a wait for I/O, followed by more processing. The natural selection of workstep boundaries was to make each processing sequence into its own workstep. The worksteps could be no larger because of the need to wait, which a workstep could not do. The only question remaining was whether the workstep was too large. This was a question of longest path execution time. In almost all cases, this natural division was not changed.

Tasks that are CPU-bound with very little synchronization with other tasks may not fit well into the workstep architecture. It would seem that some way needs to be found that allows both types of tasks to coexist.

3.2.2 Only Three Redesigns

The RTCN NOS development incorporated the execution-time estimation into the formal design/code review methodology. During the estimation and review process, only 3 of the 160 worksteps had to be redesigned as a result of estimates exceeding the limit. Two were redesigned after high-level design review and one after detail design review. None were coded and then found to exceed the time limit. It is most important to hold very firmly to that time limit.

3.2.3 Picking the Limit

Since the time limit becomes such an integral part of the development process, it becomes very important to ask how to pick the value in the first place. The key to the selection is the determination of how long the highest priority workstep can be held off and still meet its critical time requirements. Analysis of the highest speed function and of the most important function might yield this information. Remember that all functions in the processor must be considered.

The RTCN NOS team tried an educated guess; they modeled the result and selected the limit. The fastest and highest priority worksteps were concerned with moving periodic data. A small amount of work must be performed for each (no more than 50) periodic data stream during each time period. A time period can be as small as 40 milliseconds. Using this information, a preliminary guess was made that if the periodic processing was held off for as much as 2 milliseconds, or 5 percent of the smallest period, then the time constraints could still be met. A first order model was constructed using the initial guess. The model results indicated that the initial guess was sufficient, and that 5 milliseconds was borderline. The 2-millisecond limit was formally adopted, proved to be good for performance and not too restrictive for programmers. It should be noted that, at that time, 2 milliseconds represented the execution of approximately 1,000 instructions in assembly language.

3.3 Priority Scheduled

The NOS redesign addressed the issue of sharing the CPU within the network adapter (NIU). Another important issue is the sharing of the CPU in the host processor, where the application software resides. The CPU sharing question for both engines is discussed in the following paragraphs. Sharing resources goes far beyond the sharing of CPU. In fact, the identification of which resources are shared is a significant task. Also discussed are some of the other types of resources being shared in the RTCN network.

Once a resource is identified as being shared, the question arises as to how to control the sharing. As long as the resource is available, the first requester gets the resource. When the resource is not available, the new requests must be queued. The RTCN employs a priority scheme for deciding which queued request will receive control over the resource when it does become free. The priority value is derived from a parameter on each of the send data service requests.

The network could have used its own priority generation scheme but did not in order to allow the host operating system to assign priority of the I/O according to a single system wide scheme. Whether it is necessary to use a single scheme for all resources is still in question; however, it is important not to prevent the host operating system (OS) from doing so. It is necessary that the same priority generation scheme be used across the system for the RTCN send commands since these priority values will be compared to one another.

3.3.1 CPU Sharing

The NOS redesign focused attention on the method of sharing the CPU. The RTCN is designed to be concurrently working on up to 150 message-passing requests at the same time. Each request requires anywhere from 3 to 20 worksteps; a few rare requests require 500 worksteps. NOS switches control of the CPU somewhere in the range of 2,000 to 5,000 times per second. To put this in perspective, NOS executes on a 16 MHz, 68020 microprocessor with a single wait state.

Each workstep, when it is ready to execute, is represented on a priority-ordered dispatching queue. Each entry on the queue contains a workstep identifier, a pointer to the input data structure, and the priority of the work to be performed. Multiple entries on the queue will identify the same workstep, each with a different input data structure. When a workstep finishes execution, it returns control to the dispatcher, who then takes the top entry off the dispatching queue and calls the identified workstep. It is a very simple mechanism. The time consuming part of the job, maintaining the priority ordering in the queue, is aided by hardware.

3.3.2 Other Resources

One of the important lessons to be learned from RTCN is that if the CPU is scheduled for response times and the other resources are not, the weak link in the chain will show. At one point, the RTCN was working fairly well. Time-critical loads had been analyzed and were predicted to be schedulable at an 87 percent loading factor. But when the application was tested in the lab, it could not run for more than a few seconds. After careful investigation, the reason was discovered. The microcoded Bus Access Controller (BAC) was being allocated with a FIFO. It

seemed reasonable in the beginning, since the BAC handles requests very quickly compared to the 68020 microprocessor. The problem occurred when a timer started five small background tasks. One task would not have caused any deadlines to be missed, but five in a row did.

Response requirement scheduling really buys nothing when loadings are very low. Whenever a workstep needs a resource, it is available. But if loads are going to be allowed over 30 percent, something will need to be done. When is the last time anyone saw a real-time system that could afford enough hardware to keep loadings under 30 percent?

In order to help others realize just what kinds of things represent shared resources, the following list was compiled to show the shared resources, beyond CPU, found in RTCN.

- Memory Blocks
- DMA Controller
- Bus Adapter Access
- Bus Access
- Limited Number of Hardware Control Blocks

3.4 Data Driven

The concept of data-driven software architectures is not new and will be presented briefly here. If the reader is interested in obtaining more information on the subject, a search of current literature will provide all that is necessary.

Briefly, data-driven architecture refers to the characteristic that tasks, or worksteps in this case, are executed when a data structure (or a set of data structures) has reached a state that requires the processing provided by that task or workstep. Therefore, there must be a data structure that represents the state of work to be done and thereby controls the processing.

In the RTCN NOS workstep architecture, each workstep has a single data structure that contains all pertinent information for the workstep. Different worksteps may have different data structures. Some data structures contain pointers to other data structures.

The question most often asked is, "What decides that a data structure has reached a state that requires processing by a specific workstep?" The answer is simple: another workstep or the occurrence of an external event. Even though the architecture doesn't allow actual interrupts, the external events still occur.

When a data structure is ready to be processed, some software or hardware must place the workstep identifier and a pointer to the data structure on the list of dispatchable worksteps. A workstep is said to "enqueue" another workstep when it determines that the data structure is ready to be processed by that workstep.

3.5 Efficiently Dispatched

The sequence of events required to dispatch a new workstep are simple. First, the dispatcher must determine whether external events have occurred. This may be as simple as removing an interrupt mask and restoring it. Second, the dispatcher must determine which workstep has the highest priority. With hardware assist, this operation is almost as simple as reading a memory location. And third, the dispatcher must pass control to the selected workstep. To do this, the dispatcher places a pointer to the input data structure on the stack and calls the workstep entry point as a subroutine.

3.6 Extension to other tasking models

There may be other tasking models that also exhibit the guaranteed response-time characteristics and may be better suited to CPU-bound tasks. One suggestion that was made is to allow a coded task to ignore the time limit and to have the OS enforce the time limitation via watchdog timer. Many of the beneficial workstep attributes, such as the limited number of states, are eliminated by this approach, but it may still have merit.

4.0 Dynamic Memory Management

Another key to sharing resources within the NOS microprocessor is memory management. The dynamic management of data storage was identified as one of the central contributing factors within the serious performance problem. Each task had an area of memory allocated to the task throughout its existence called static memory. In addition, as extra memory was needed, it was obtained temporarily from the OS; this was called dynamic memory. In the old architecture, the static areas were small, while most memory was obtained dynamically. The amount of CPU time spent in the OS managing memory clearly indicated the need to change. Unfortunately, there was insufficient memory to allocate all data as static memory. A judicious blend of static and dynamic memory management was indicated.

The method selected has been used by other OS. A program (i.e., a set of tasks) is statically allocated memory to be used as dynamic memory pools of fixed size memory blocks. The dynamic memory pools are shared between tasks of the same program but not between different programs in the same processor. The allocation of static memory and dynamic pools is done at the time the program is bound to the OS. This was done at compile time for the NOS, but there is no reason that this could not have been done at program load time. The primary concern is that the OS be provided knowledge of the program's dynamic memory use before program execution begins.

Another concern is that the programmer be held accountable only for the effects of his own program. If a programmer creates program "A" that correctly executes when tested by itself, then the execution must be correct when the program is sharing a CPU with another computer. If the two programs are allocated dynamic memory out of the same pool, then program "A" may not exhibit the same concurrency when sharing as when it was tested by itself. Why should the programmer of "A" be held accountable? The problem is with the OS not program "A."

If multiple programs are to share a limited resource, each must have an enforced limit to its own use. The program should state its limits, should not be loaded unless its needs can be supplied, and the OS should enforce those limits. Only in this way can each program remain independent.

4.1 Allocation of Resource Pools

The previous section explains how memory was subdivided into usable blocks of static sizes. The method of low overhead allocation and collection still remains a problem. The answer is to use the hardware FIFOs. A queue of free blocks was created at initialization time and maintained in the hardware FIFOs. The only problem is that a hardware FIFO element is not large enough to contain a pointer

(32 bit address). The solution is to allocate an array of memory blocks and to write the index of free blocks on the hardware FIFO.

4.2 Analysis of Pools

The problem of "How many blocks are sufficient?" seems to be the largest stumbling block for most programmers. However, if one considers the problem of dynamic memory allocation under a generalized OS, the question remains and sometimes programmers tend to leave the question to the OS. The programmers of the operating system have no way to determine how much is enough, so they make the remainder of memory available. Often the question is never answered until the application breaks down due to insufficient memory.

Software designs must analyze the amount of allowable concurrency and must develop strategies and code to react properly when the limit is exceeded. The analysis of concurrency will also answer the question of "How many blocks are sufficient?"

5.0 Modeling

Although models were created during the development of the original RTCN NOS, they were not the focal point of the development activity as they were during the redesign activity. Models were created that served throughout the design and development process as guides to aid in the decision making incumbent in that process. In order to be useful, a model must be easily modified to show the results of different design choices. Making a model simple enough to modify easily is strongly influenced by the tasking model. The workstep architecture makes the design and use of models both accurate and simple to modify.

5.1 Workstep Execution Attributes

Each workstep represents a discrete event in time that consumes resources and takes a predictable amount of time. Sometimes the amount of time taken is a function of a loop parameter. However, because of the design constraints, a maximum time exists. In addition, the function of the dispatcher is directly described in a model as a queue of events. These attributes combine to make the workstep architecture easy to model.

Sequences of worksteps are serially related to create a function description, which are then described in a scenario description. Each workstep must be described by the amount of time it takes and the specific resources that it consumes. Only resources that need to be modeled must be described. That is, if the model must aid in the selection of memory block allocation queue depths, each workstep description must include the number of memory blocks acquired and released.

5.2 Workstep Modeling Levels

Two kinds of models were created for RTCN NOS. The first was called a "First Order Model" because it could not take into account the delays encountered due to resource contention. The "Second Order Model" computes resource contention delays.

5.2.1 First Order Model

The First Order Model was created on an IBM PC using a popular spreadsheet program. Spreadsheet models have the advantage of giving results very quickly. Each major function of NOS has a spreadsheet model to describe the response time and resource usage of the function. Functions are combined in scenarios and total CPU-usage levels are computed for those scenarios.

This model is easily modified and gives a "quick look" at the resources for a specific combination of functions. The combination of functions that are of the greatest interest are the functions called out in the performance specification. These combinations constitute the basis of any acceptance test and measure the success of the software design. The combinations of functions stated in the functional specification were modeled to determine whether specified performance levels would be obtained. Once the original estimates were modeled and showed a comfortable margin for growth during development, the estimates were declared to be resource budgets. Before any request for more CPU or other resources was granted, the request had to be added to the model and show that all specified performance levels would still be provided.

More recent models of a workstep architecture have varied the approach to allow a spreadsheet model of a single Bus Interface Unit (BIU) to include delays due to contention for resources within the single BIU.

5.2.2 Second Order Model

The Second Order Model is written in a current simulation language to provide a more in-depth view of the network action. While it takes longer to create and manipulate, the second order model is a discrete event simulation that fits very nicely with the fact that the workstep architecture takes action on discrete events. The simulation can encompass more than the spreadsheet models; that is, more than two nodes, bus resource, and bridges.

The spreadsheet models give results such as the average delay, average queue depth, and average response time. By calculating the effect of each event, the simulation creates actual results and then can show the distribution of delays, queue depths, and response times. The simulation can show the effect of priority; where the model must assume that all commands are of the same priority.

The simulation was not used as extensively as the model because the changes in the workstep relationships were harder to reflect in the simulation language than in the spreadsheet. Most "what if" questions were answered by the spreadsheet model. The simulation was updated at each major milestone in the software development. Simulations of both the scenarios representing specified functional performance and those representing anticipated use of the network were run after the updates and when significant changes were made to the anticipated-use scenarios.

5.3 Modeling results

Model and simulation results were both early and significant. The original guess at the maximum workstep execution time was 2 milliseconds. (A half millisecond was clearly too low and 10 milliseconds was too high.) When the model, and later the simulation, were set up with the original assumptions based on this maximum, the results indicated two things. One, that the 2-millisecond limit was sufficiently small to allow high-priority work to complete on time. Second, that performance levels of twice the specified values should be achieved. These results were indicated as early as six weeks into the year-long effort to redesign NOS.

It should be noted that the final test of NOS showed performance levels of 178 percent of specified values using just 43 percent of the CPU. The test did not test

higher levels due to the fact that the test driver computer had reached full CPU load. The simulation had shown that the 200 percent performance level would be reached with 80 percent of the CPU. However, all worksteps were modeled with their worst-case execution path, introducing a bias toward the high side.

Model and simulation results are never any better than the quality of the assumptions upon which they are built. Lead software and system engineering personnel must maintain the highest degree of quality in the model and simulation assumptions throughout the software development process. The key to quality assumptions is to provide a tasking model that allows programmers to employ the assumptions as design constraints. Programmers should not be alienated by being held responsible for things that are outside their control. A programmer can tell how long a workstep will take to execute but not how long before the external event occurs that allows the workstep to begin execution. In addition, the programmer cannot tell how long a workstep will wait for the CPU while higher priority worksteps take control.

6.0 Hardware Assist

Real-time systems must be efficient and predictable. It is the job of the OS to control the sharing of resources within a processor. The OS must, therefore, share those resources in an efficient way that results in predictable behavior. Response times, which are the key to real-time systems, must be guaranteed. These guarantees are achieved through resource allocation based on response-time requirements. In order to allocate resources in this way, the OS deals with queue operations, timed events, and interrupt handling. The RTCN experience shows that it is possible to augment the OS with special-purpose hardware in order to create a highly efficient OS with response-time guarantees.

Hardware assistance for prioritized queues and FIFOs as special-purpose memory locations reduced the largest OS overhead. The workstep dispatching function was reduced from 500 to 700 microseconds to less than 5 microseconds through the use of a hardware priority queue. The starting and stopping of timed events was simplified by using memory-mapped hardware-decrementing counters. FIFO queues are used for allocation of available resources. Priority queues are used when available resources are depleted and resource requestors must wait for access.

The concept of data-driven worksteps was also reflected in the interface between NOS and the hardware portions of the NIU. Hardware interfaces were developed as command/response functions. The command issued by a NOS workstep to a piece of NIU hardware would contain the information needed to write the memory-mapped, workstep-dispatch queue. When the hardware had finished its job, it would simply write a memory location that would eventually cause the responding workstep to be executed. When hardware interfaces are designed this way, no interrupts occur to signal device completions. The essence of simplicity is using the construct of writing a pointer into a memory location causing a hardware function to be invoked.

7.0 Conclusion

This experience is at a minimum an interesting data point for discussion. At a maximum, it may contain the basis for some new approaches to real-time OS. It raises questions concerning the trend to "all-knowing" OS. It asks that programmers be held accountable for knowing the resources that they will require and for providing the OS with sufficient knowledge to allow deterministic management of those resources. It asks that tasking models not be concerned with "fairness" but with the ability to guarantee response times. It asks that development techniques be improved to aid in the management of response times with models early in software development.

Real-Time Control of an Autonomous Land Vehicle

by

Jon McSwain
and
Tom Richardson

Martin Marietta I&CS
PO BOX 1260
Denver, CO 80201-1260

1. INTRODUCTION - PROJECT DESCRIPTION

Martin Marietta Information & Communication Systems in Denver is currently working on an Autonomous Land Vehicle (ALV)[1].[1] The contract has developed a robotic vehicle mobility test bed. The focus of the project has been to:

(1) Perform the R&D tasks necessary to develop a computer controlled vehicle.

(2) Benchmark the state of the art in Advanced Computer Architectures, Vision Processing, and Artificial Intelligence through public demonstrations of the ALV.

(3) Make the ALV available to the robotic research community in support of their experiments.

During the first phase of the project, from 2/85 to 2/88, the demonstrations focused around autonomous, computer controlled road following. Areas of research contributing to these demonstrations were: computer vision to perceive the road, path planning to describe a safe path for the vehicle to follow, and real time control to execute the commands of the path planner.

The final system engineered by Martin Marietta holds the record for high speed autonomous road following. This system is capable of driving the vehicle at an average speed of 14 kilometers per hour (kph), over a 4 kilometer test track that included 2 hair-pin turns, several more gradual bends in the road, hills, and intersections. Top speed during this demonstration was 20 kph. Two obstacle fields comprised of static obstacles were negotiated.

2. VEHICLE DESCRIPTION

The vehicle consists of an eight wheeled skid steer platform built by Standard Manufacturing Company. The wheels are powered by two hydraulic motors that are in turn powered by a diesel engine. Top speed of the vehicle is currently around 25 kilometers per hour on level ground. Electric power for the vehicle's computers is supplied by an on-board 30 kilowatt diesel powered generator. Cooling for the computers is provided by a 100,000 BTU air conditioning system driven by the main engine.

---

1 - This work was performed under Contract Number DACA76-84-C-0005 of the Autonomous Land Vehicle Program sponsored by the Defense Advanced Research Projects Agency (DARPA) as part of its Strategic Computing Program and contracted through the U.S. Army Engineer Topographic Laboratories.

The vehicle is designed to provide a mobile computer room. The inside of the vehicle is spacious enough to allow 2 to 3 engineers inside to operate the on-board computers.

Human control of the vehicle is accomplished by a radio control link not unlike those of model cars. There are no windows or vehicle mobility controls inside the vehicle. Dead-man control, eg. emergency stopping, of the vehicle is accomplished by a series of switches on the same radio control box.

3. HARDWARE AND SOFTWARE ENVIRONMENT

Computer equipment on-board the vehicle is shown in *Figure 1* and consists of:

- o 3 SUN 3/180 work stations, two with graphics terminals,
- o 2 Vicom 1800 image processors,
- o A Multibus card cage supporting approximately 5 Intel processors,
- o A General Electric Warp systolic array processor,
- o A gateway to the lab.

Also shown in the figure, but not on-board the vehicle, is a radio link. The link has a full duplex digital link at T1 (1.56 mbps) and 56kbps rates and four video links from the ALV to the lab. Using the networking software described below, this subsystem allows access to the ALV laboratory systems. Neither the lab equipment nor the radio link was used in the Martin Marietta demonstrations. The radio link has been used by the Hughes Artificial Intelligence (AI) center (California) during their ALV experiments in Denver. The lab systems include:

- o Two Symbolics Lisp machines,
- o A 16 node BBN Butterfly MIMD parallel processor,
- o A second GE Warp ,
- o Several more SUN systems,
- o An Intel system,
- o The gateway for the communications to the ALV.

Sensor equipment used to accomplish the phase one demonstrations consisted of:

- o An RCA CCD color television camera,
- o A laser radar system built by Environmental Research Institute of Michigan (ERIM),
- o A Bendix inertial/odometric land navigation system (LNS).

Vehicle software environments consist of Unix for the SUNs, Versados for the Vicoms, and iRMX for the Multibus system.

Computer control of the vehicle is accomplished by a custom interface designed and built by Martin Marietta. This interface is connected at one end to the Multibus card cage, and

at the other to the control inputs of the hydraulic motors. Additionally, several discrete functions, for example turning the brakes off, or selecting high gear, are controlled from the Multibus system via the vehicle interface board.

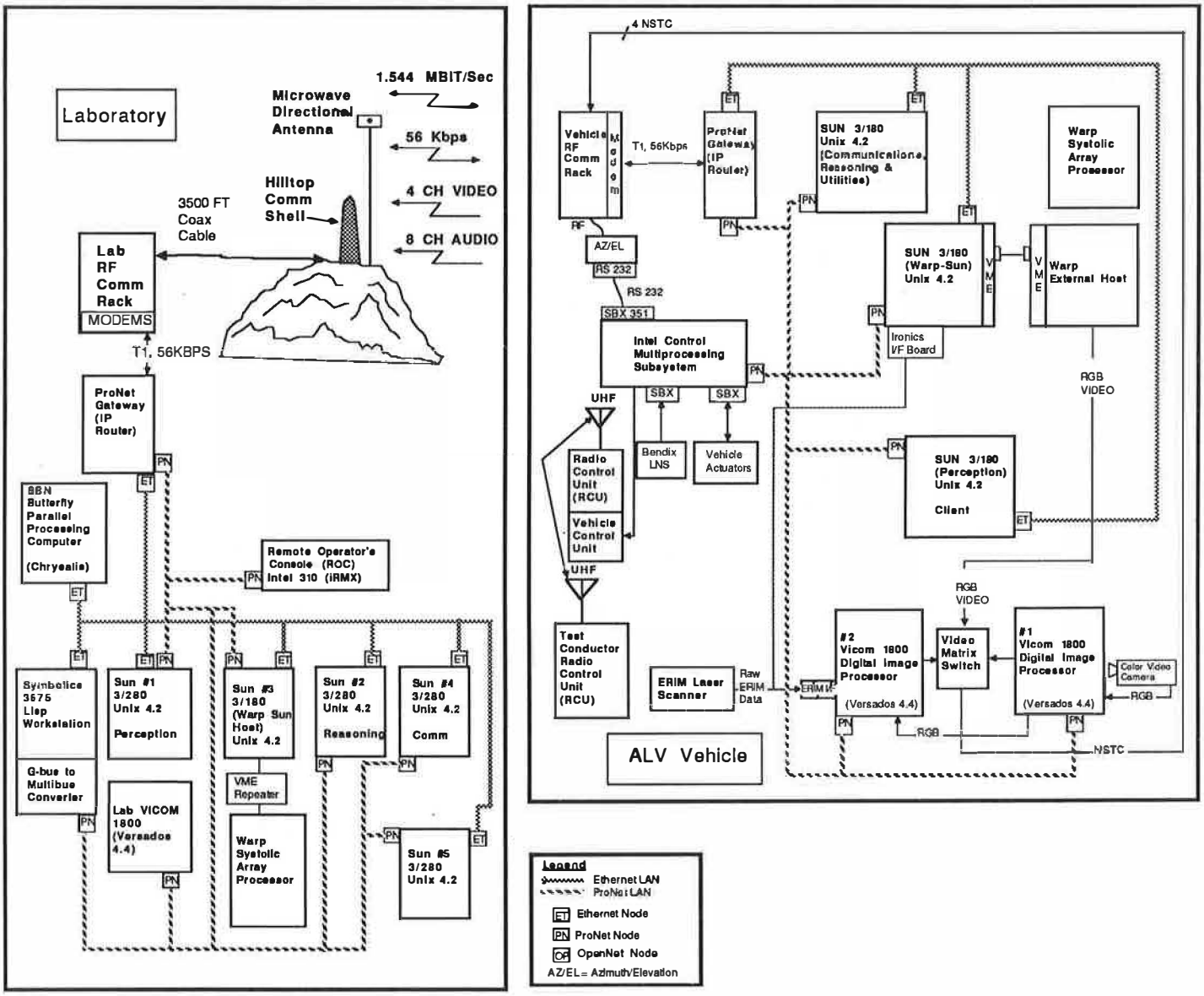


Figure 1: ALV System Hardware Architecture

4. ALV LOCAL AREA NETWORK -- *RealNet* ™

The heterogeneous computer systems are linked together by a token ring capable of 80 megabits per second. The hardware is manufactured by the Proteon Corporation. The device driver software and application interface software were developed by Martin Marietta.[2],[3] The software is currently being marketed [2] by Martin Marietta under the

name *RealNet* ™.

4.1. User Interface

Data transfer between applications is accomplished by three function calls: **Send, Receive,** and **Status.** Data record routing information is contained in 4 configuration files that are read by the system during network initialization.

4.2. Description Of Data

Any process may originate an item of data, but only one originator is allowed per data item. Any process may serve as the destination of any data except data items that it originates.

The processes do not need to know the proper internal data representation of a particular host. *RealNet* ™ automatically resolves byte/swap and word/swap reformating.

There are two separate classes of data used on *RealNet* ™. They are **Refresh** and **Multiple Occurrence**. Refresh data is used when the application is concerned with only the most recent data. **Multiple Occurrence** data is used when a history of the data is required, i.e., all the data is necessary.

Refresh data items have only two buffers assigned for the receipt of data. The two are used to prevent the inadvertent overwrite of data being read if more data is received. The old data is overwritten each time new data arrives. In this manner, only the most recent data is available to be read.

Multiple Occurrence data items are handled differently. Because all the data is required, a larger set of buffers is allocated to **Multiple Occurrence** data items. The structure of the buffers is a revolving FIFO and the oldest item is overwritten once the buffers are all full. The numbers of buffers can be adjusted by the user. In the case where several processes on the same host are to receive the data, a particular buffer will not be reused until all the processes have received it. If a process does not receive the data before the buffer becomes the oldest and has to be reused, the loss of data is recorded in the status file for the host.

4.3. Results

The implementation of this networking system allowed the record setting ALV demonstration to take place. Other networking schemes were either too slow and non-deterministic to allow the ALV to meet its performance requirements (i.e. Ethernet), or were too homogeneous to be practical (i.e. a DMA scheme). In actual tests from application process to application process on the Sun 3/180 work stations, a seven fold increase over socket communications was realized.

5. ALV APPLICATION ARCHITECTURE

The ALV software system is a hierarchical architecture, where levels of activity were defined based on specific tasks' differing levels of immediacy. The lowest level concerned itself with controlling the vehicle's motors. Software executed on the Multibus system under Intel's iRMX operating system was responsible for the real-time control of the vehicle. The control software was event driven by the LNS update rate that caused a position update to arrive from the LNS processor every 40 milliseconds.

The next level of activity was event driven by the minimum processing time of the computer **Perception** system.[2] Using the **Perception** scheme described in the next paragraph, a complete model of the road was generated about every second. A path planner employing a potential fields algorithm would then produce a path for the control system to execute about one half second after receipt of a road model.

---

2 - The commercialization effort for *RealNet* ™ is being performed using Marietta Marietta provided funding.

In concurrent operation were two different **Perception** tasks. The first was concerned with modeling the blacktop road. This system employed the video camera, two Vicom image processors running the same software 180 degrees out of phase, and one SUN.[4] The second task was concerned with detecting and modeling obstacles.[5] This system ran on the GE Warp systolic array processor with a SUN host. It used the ERIM laser radar for sensor input. The path planning system was event driven from either of these two inputs. Sensor fusion of these two sources was accomplished as needed.[6]

The highest level of activity was the execution of the mission plan. This consisted of global coordinates paired with actions. Actions were typically "stop", or "turn around". These activities occurred when the vehicle arrived at the designated point.

6. ALV APPLICATIONS

The ALV system has accomplished three large scale demonstrations. The last of these occurred in November of 1987, and is reviewed in the introduction. Other uses of the vehicle include computer vision research by University of Maryland[7], SRI[8], and Advanced Decision Systems. Hughes AI center is currently conducting research for off-road autonomous mobility using the ALV. They have designed and implemented their own software system, using only the Martin Marietta Multibus control system software, the *RealNet* ™ Local Area Network (LAN), and the high speed radio link to the ALV computer lab.[9]

During the next two year contract, emphasis of the program is to allow more organizations outside of Martin Marietta access to the vehicle. Anyone interested should contact Dr. Robert Douglass at either 303-977-5159 or through email at douglass@martin-den.arpa. For Further information about *RealNet* ™ contact Tom Richardson at (303) 977-6268.

7. References

[1] "Strategic Computing - New Generation Computing Technology: Strategic Plan for Its Development and Application to Critical Problems in Defense," Defense Advanced Research Projects Agency, October 1985.

[2] "RealNet Design," The Autonomous Land Vehicle (ALV) Fourth Quarterly Report, February, 1987.

[3] Holland, R., Gothard, B., Celvi, G., Idea Report #87YD23 Martin Marietta, "RealNet Real Time Local Area Network Software" May, 1987

[4] Turk, M.A., Morgenthaler, D.G., Gremban, K.D., and Marra, M., "VITS -- A Vision System for Autonomous Land Vehicle Navigation," To be published in IEEE PAMI, 1988.

[5] Dunlay, R.T., Hennessy, S.J., and Morgentahler, D.G., "Obstacle Avoidance for the Autonomous Land Vehicle: Perception," Unmanned Systems, Summer, 1987.

[6] Dunlay, R. T., "Obstacle Avoidance Perception Processing for the Autonomous Land Vehicle," Proceedings of the IEEE Conference on Robotics and Automation, Philadelphia, PA, April, 1988.

[7] DeMenthon, D., "Inverse Perspective of a Road from a Single Image," Technical Report CAR-TR-210, Center for Automation Research, University of Maryland, July, 1986.

[8] Lawton, D., Levitt, T., McConnel, C., Nelson, P., Glicksman, J., "Environmental Modeling and Recognition for an Autonomous Land Vehicle", Proceedings of the Image Understanding Workshop, February, 1987

[9] Daily, M., Harris, J., Keirsey, D., Olin, K., Payton, D., Reiser, K., Rosenblatt, J., Tseng, D., and Wong, V., "Autonomous Cross-Country Navigation with the ALV," Proceedings of the DARPA Knowledge-Based Planning Workshop, Austin, TX, December, 1987.

SESSION VI

REAL-TIME DATABASES

Chairman

Kang Shin
University of Michigan

Scheduling Hard Real–Time Transactions

Jane W. S. Liu, Kwei–Jay Lin and X. Song
Department of Computer Science
University of Illinois
1304 West Springfield Avenue
Urbana, Illinois 61801

Real–time processes often share data. A concurrency control mechanism must be used to ensure data consistency, making it difficult to schedule such processes to meet timing constraints. In general terms, this is the problem of scheduling hard real–time jobs subject to resource constraints [1–6]. It has been shown that this problem is NP–complete. Several heuristic algorithms for scheduling jobs subject to resource constraints have been developed. These algorithms are mostly for the case when there are a small number (for example, 10 or 20) of resources and are not suitable when the resources, such as data, are numerous (for example, 1,000 or more). For many applications, the problem of scheduling jobs that share data can be formulated as that of scheduling conflicting transactions. In this formulation, computations are modeled as transactions. Each transaction $\tau = s_1, s_2, \cdots, s_k$ is a sequence of steps s_i, $1 \leq i \leq k$, reading or writing the values of (data) objects in a database.

We approach this problem from two different directions. On the one hand, we take the traditional approach in analyzing the schedulability of conflicting transactions and finding suboptimal heuristic algorithms to schedule them. On the other hand, we adopt a new approach by introducing a set of continuous criteria for temporal consistency that is more appropriate for many hard real–time applications. This paper discusses these two approaches and our work in this area.

The Traditional Approach

In the traditional approach, the database state at any time is specified by the values of all objects in the database. The state is said to be consistent if the values satisfy a prescribed set of integrity constraints. Every transaction is assumed to be correct in the sense that it transforms the database from one consistent state to another consistent state when executed alone. Let $T = \{\tau_1, \tau_2, \cdots, \tau_n\}$ be a set of transactions. A *schedule* σ for T is a rearrangement of the steps of the transactions $\tau_1, \tau_2, \cdots, \tau_n$ such that if s_j and s_k are steps of the transaction τ_i with $j < k$, then s_j precedes s_k in the schedule σ. A schedule generally specifies a concurrent (or interleaved) execution of the transactions in T. A concurrent (or interleaved) execution of T is said to be *state–serializable* if the final state of the database resultant from this execution is the same as one resultant from a serial execution of the transactions, and is said to be *view–serializable* if the values read by every transaction is the same as the values read by it in a serial execution [7–11]. A schedule is *correct* if the resultant concurrent execution is serializable in one of these senses. The difficulties in scheduling hard real–time transactions are elaborated in [12–14].

We have been concerned with the schedulability of aperiodic transactions that have the same ready time and deadline, use a locking protocol for concurrency control, and run on a multiprocessor system. To study this problem, we model each transaction in a system T of n

transactions as a sequence of steps with each step s_i being either a read of an object x, a write of x, a $lock(x)$ for exclusive access, or an $unlock(x)$ to releases x. We have shown that the problem of determining whether the transactions in T can be completed on p processors at or before their deadline d can be formulated as that of determining whether a path of length d or shorter with some special properties exists in a n–dimensional cube. This formulation is based on the geometric representation used for studying concurrency control by locking [8]. It provides us with a method that can be used at compile time to determine the schedulability of transactions. Schedulability analysis has been done in an ad hoc manner in most known works. Typically, the fact that the algorithm used fails to find a feasible schedule does not means that a feasible schedule does not exist. Determining the schedulability of transactions is further complicated by the fact that the concurrency control mechanism may allow nondeterminism in the order of their executions and hence more flexibility in scheduling them. Most known schedulability analysis algorithms cannot handle this situation effectively.

We illustrate our schedulability analysis method by the simple example shown in Figure 1: there are two transactions, τ_1 and τ_2, to be executed under the control of two–phase locking protocol. Both of them read and/or write objects x and y and are well–formed, that is each read or write of an object is preceded by a *lock* and followed by an *unlock* of that object. Each transaction comprises of eight steps. For the moment, we let every step take 1 unit time. Figure 1 shows a 2–dimensional plane in which the two axes represent the two transactions and the grid points (i.e. the intersections of the horizontal and vertical lines) on the axes represent the steps in the transactions. Only steps involving locking and unlocking are shown. A point in this plane represents a state of progress towards the completion of the transactions, from (0,0), where no

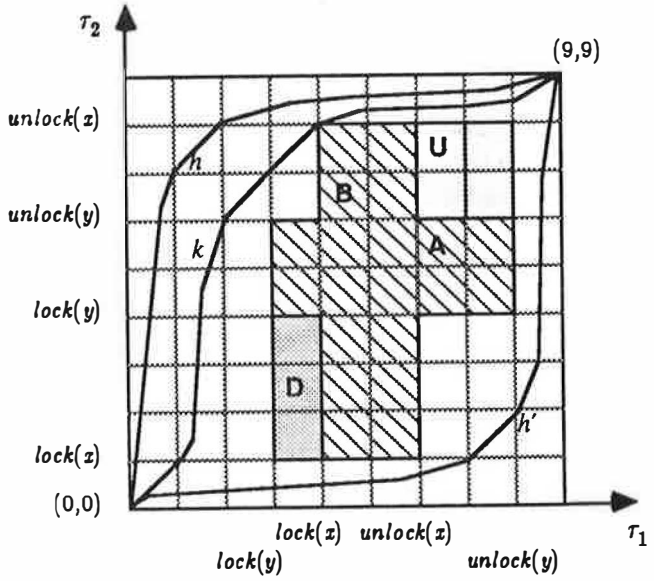


Figure 1. An example: A 2–transaction system with 2–phase locking

step is executed, to (9,9), where all steps are completed. The shaded area marked A and B is called the forbidden region since it contains points representing unreachable states. The area marked D represents the deadlock region containing states that will lead to deadlock or are deadlock states. A state–serializable, deadlock–free schedule is represented by a non–decreasing path from (0,0) to (9,9) outside of the forbidden region and the deadlock region [8]. Figure 1 shows three such paths marked h, h', and k. A serial schedule in which the steps in τ_1 are executed before the steps in τ_2 is represented by a path which runs first parallel to the horizontal axis and then to the vertical axis.

We generalize this geometric representation to describe the execution of the transactions on a 2–processor system as follows: The length of a path is equal to the total number of horizontal or vertical grid lines it crosses plus the number of grid points it crosses, not including (0,0) and (9,9). Each crossing of a horizontal (or vertical) grid line by a path represents a step of τ_2 (or τ_1) executed on one of the processor by itself with the other processor left idle. A diagonal crossing of a grid point represents a parallel execution of two steps, one of each transaction. The completion time of a schedule represented by a path is equal to its length. In this example, the length of the two paths (not shown) representing serial schedules is 16 since each of them crosses 16 grid lines and no grid points except (0,0) and (9,9). The lengths of the paths h, h' and k are 14, 14 and 12, respectively. They represent three correct, concurrent schedules in which 2, 2 and 4 steps of τ_1 are executed in parallel with steps of τ_2. These schedules have completion times 14, 14, and 12, respectively. It is can shown that there exists no non–decreasing path outside of the forbidden region and the deadlock region from (0,0) to (9,9) of length less than 12 and, hence, no schedule with completion time less than 12.

The representation illustrated by Figure 1 can be used to represent transactions whose read and write steps take unequal time. In this case, the width of the grid lines represents a slice of time chosen so that a lock or unlock step can be done in one time slice. A path such as the ones shown here represents a preemptive schedule in which preemption is allowed only at the end of every time slice.

To determine whether a set of n transactions can be completed by time d on p processors when preemption is allowed, we can represent the transactions as a n–dimensional cube. Given the locking protocol used, the forbidden region and the deadlock region can be easily computed. It has been shown in [8] that a locking protocol ensures state–serializability if and if only the forbidden region (referred to as the closure of the union of the forbidden rectangles in [8]) is connected. A non–decreasing path representing a correct, deadlock free, preemptive schedule on a p–processor system is parallel to all except at most p axes of the cube and is outside of the forbidden region and the deadlock region. A general algorithm for finding shortest paths in a graph can be applied to find shortest such paths, i.e. schedules with the minimal completion time. The general algorithm is exponential in n since the number of points in the cube is exponential in n. Because of the regularity in our geometry, however, it is not necessary to examine every point in the cube in the search for a shortest path, as the general algorithm does. We are currently evaluating an efficient algorithm which examines only the corners of the forbidden region and the deadlock region. In particular, we want to determine the chance for existing shortest paths being missed in the search and the worst case performance in terms of the length of the shortest path found by the algorithm.

Another problem being addressed is that of scheduling transactions with preallocation of resources and arbitrary ready times and deadlines. In this case, conflicting transactions are constrained to be executed sequentially. Hence, one way to schedule these transactions is to first generate a precedence graph from the given conflict graph and then apply known efficient algorithms to find feasible schedules from the precedence graph. We are currently evaluating a class of heuristic algorithms to generate and choose valid precedence graphs that are consistent with the given conflict graph and are likely to yield feasible schedules if they exist.

Temporal Consistency

We also take an approach that is complementary to the traditional approach to scheduling hard real–time transactions. In many applications where timing is critical, the notions of state– and view–consistency traditionally used in concurrency control studies can sometimes be replaced or supplemented by the notion of temporal consistency. Specifically, we consider database objects in a real–time database as models of real–world objects. For example, consider a system in which input sensors read the values of real–world objects and the resultant sensor inputs are written into the database. For every real–time object X_r there is a database object X, called the *image object*, or simply image, of X_r. The value of X is equal to the sampled value of X_r taken at the last sample instant. (To simplify our discussion here, we neglect error in the clock.) There are database objects that are not images of any real–world object. *Derived objects*, whose values are computed from values of image objects, are in this class as well as *constant objects*, whose values are constant independent of the values of real–world objects and time. The *age* of any real–world object or constant object is zero at all times. The absolute age at time t of a image object X is $t-t_l$ where t_l is the time at which the corresponding sensor was read and the current value of the object was defined (that is the time elapsed since the last sample instant.) The age of a derived object Y whose values is computed from the values of the objects in the set $\{X_i\}$ is equal to the maximum age of these objects plus the time spent to generate the value of Y and write Y. A set $\{X_i\}$ of objects is said to be in a *absolute temporal consistent* state at t if the ages of X_i are equal to or less than a certain given threshold A. The *dispersion* (in ages) of two objects is the difference in their ages. A set $\{X_i\}$ of objects is said to be in a *relative temporal consistent* state if their dispersion is equal to or less than a threshold R. The normalized age and dispersion with respect to A or R are called the absolute and relative *incoherence*, respectively.

An objective of a concurrency control mechanism in a hard real–time system is to keep the incoherence of supposedly temporally consistent objects equal to or less than 1 in addition to maintaining state–consistency or view–consistency whenever it is necessary. Existing concurrent control algorithms for typically hard real–time applications should be evaluated to determine their performance in term of average and worst case incoherence and the probability of objects being temporally inconsistent. We suspect that while optimistic concurrency algorithms have better response time, they are likely to have to higher worst case incoherence compared with some pessimistic concurrency algorithms. We are evaluating several commonly used concurrency control algorithms for this purpose.

References

[1] Blazewics, J., J. K. Lenstra, and A. H. G. Rinnooy Kan, "Scheduling subject to resource constraints: Classification and complexity," *Disc. Applied Math.*, Vol. 5, pp. 11–24, 1983.

[2] Garey, M. R., R. L. Graham, D. S. Johnson, and A. C–C Yao. "Resource constrained scheduling as generalized bin packing", *J. Combinatorial Theory*, V. 21, pp. 257–298, 1976.

[3] Zhao, W., K. Ramamritham, and J. A. Stankovic, "Preemptive scheduling under time and resource constraints," *IEEE Transactions on Computers*, pp. 949–960, August 1987.

[4] Zhao, W., K. Ramamritham, and J. A. Stankovic, "Scheduling tasks with resource requirements in hard real–time systems," *IEEE Transaction on Software Engineering*, April 1985.

[5] Mok, A. K., "Fundamental design problems of distributed systems for the hard real–time environment," Ph.D. Thesis, Department of Electrical Engineering and Computer Science, M.I.T., 1983.

[6] Sha, L., R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols — an approach to real–time synchronization," Technical Report No. CMU–CS–87–181, Carnegie Mellon University, November 1987.

[7] Berstein, P. A., D. W. Shipman, and W. S. Wong, "Formal aspects of serializability in database concurrency control," *IEEE Transactions on Software Engineering*, Vol. SE–5, No. 3, pp. 203–215, 1979.

[8] Papadimitrious, C. H., "Concurrency control by locking," *SIAM Journal on Computing*, Vol. 12, No. 2, pp. 215–226, May 1983.

[9] Ibaraki, T., T. Kameda, and T. Minoura, "Disjoint–interval topological sort: a useful concept in serializability theory," *Proceedings of the 9th VLDB*, pp. 89–91, Florence, Italy, 1983.

[10] Yannakakis, M., "Serializability by locking," *Journal of ACM*, Vol. 31, pp. 227–244, 1984.

[11] Vidyasankar, K., "Generalized theory of serializability," *Acta Informatica*, Vol. 24, pp. 105–119, 1987.

[12] Sha, L., J. P. Lehoczky, and H. Tokuda, "Real–time network scheduling and real–time database theory," *Proceedings of the Real–Time Systems Issue Workshop*, Austin, Taxas, November 1986.

[13] Sha, L., J. P. Lehoczky, and E. D. Jensen, "Modular concurrency control and failure recovery: part I: concurrency control," to appear in *IEEE Transactions on Computers*

[14] Sha, L., J. P. Lehoczky, and E. D. Jensen, "Modular concurrency control and failure recovery: part II: failure recovery," to appear in *IEEE Transactions on Computers.*

Partial Computation in Real-Time Database Systems

Susan B. Davidson and Aaron Watters
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

March 1, 1988

Abstract

A critical component of real-time systems is the database, which is used to store external input such as environmental readings from sensors, as well as system information. Typically, these databases are large, due to vast quantities of historical data, and are distributed, due to the distributed topology of the devices controlling the application. Hence, sophisticated database management systems are needed. However, most of the time the databases systems are hand-coded. Off-the-shelf database management systems are not used due in part to a lack of predictability of response [1,2]. We motivate the use of partial computation of database queries as a method of improving the fault-tolerance and predictability of response in real-time database systems.

1 Introduction and Motivation

Real-time systems define correctness as providing the correct result in a timely manner. If the timing requirements for a computation cannot be met, the computation fails. To relax this definition of correctness and somehow *tolerate* the failure of meeting a deadline, one must either be willing to accept the results of computation late, or be willing to accept partial, poorer quality results in a timely manner. The first strategy interprets timing constraints as being "soft": the completion of a computation or set of computations has a value to the system which is expressed as a function of time. The system schedules these computations to maximize the total value to the system; however, it does not guarantee that all computations will be performed at their local maximum value [3]. The second strategy requires the computations to have an iterative or multi-phase nature. Furthermore, they should be *monotonic* in the sense that "goodness" of the answer is monotonically non-decreasing as the computation proceeds [4,5]. An example of such a computation is the bisection method for finding a root of a function: The interval containing a root is initially very large, and keeps halving as the computation proceeds. At any point in the computation, the interval is valid; however, it is best defined when the endpoints of the interval converge to a single point, the root of the function.

While "soft" timing constraints have recently been proposed for transaction management in real-time database systems [6], little work has been done on generating *partial* or iterative answers to queries. However, we feel that the ability to give such answers is extremely important since the expected execution time of a query is very difficult to predict. In distributed databases (which is typically how real-time databases are structured), the unknown size of the relations to be transmitted makes it difficult to bound communication time; long-lived communication failures can also cause indeterminate delays. Indeterminate delays can also result from locking due to concurrency control in the local (centralized) databases. Unfortunately, in traditional databases systems, unless *all* structures necessary to answer the query are accessible, there is no notion of an answer. Our notion of partial computation for queries is therefore a "best-estimate" of the final answer, based on *the structures that are currently available.* Our interpretation of "monotonicity" is that any fact which is said to be true remains true as computation proceeds, and any fact which is said to be false remains false as computation proceeds.

As an example of how a partial answer to a database query could be useful, suppose that we have a distributed sytem of three blood bank databases. Each blood bank database maintains, among other information, a relation of how many pints of each blood type is currently on hand. Suppose that type O- is dangerously low at hospital X, and X is trying to find out if there is any available within the network of blood banks to meet a current crisis. This query could be expressed as: "Is there a blood bank that has blood of type O-?", Thus, the query "Do you have blood of type O- ?" would be broadcast to each of the three databases, and the final answer would be the logical "or" of the responses from each of the databases. The initial partial anwer to the query would be "I don't know yet." This partial answer can be changed to "Yes" immediately some blood bank responds with a "Yes", regardless of whether all blood banks have responded. The answer becomes "No" only when all of the blood banks have responded negatively. However, at any point in time, there is some answer to the query that is correct. If hospital X then wanted to know "How much blood of type O- is there in the system?", the initial partial answer would be "At least 0", and would be improved by adding the total amount from each database as the information became available. For instance, if the first blood bank responded with "10 pints", the answer would be improved to "At least 10 pints." If the second and third blood banks responded with 5 and 25 pints respectively, the answer would become "Exactly 40 pints."

This example illustrates several points:

- It is an example of a real-time process in which the response must be predictable: Hospital X cannot wait indefinitely for an answer from the blood banks since in the worst case that there is no O- blood it must start rounding up donors to cover any anticipated crisis. Note,

however, that the timing constraint is probably minutes (or hours, depending on how nervous hospital X is) rather than milliseconds.

- A "hand-coded" query system which anticipates this type of query probably would act as in the example, while a strict relational algebra system would not be optimized to give partial information.

- The "goodness" of the answer given to the user is monotonically non-decreasing with time. Given a partial answer "At least 10 pints." the user can infer that the total is *definitely not* less than 10 pints, and *possibly* any integer greater than or equal to 10 pints (11 or 1198, for example). Furthermore, the answer given at an earlier stage is never contradicted at a later stage.

2 Partial queries

The reason why conventional query languages (and the relational algebra in particular) do not seem to be amenable to an iterative method is that the relationship of individual relations (or whatever structure is used in the model) to the final result is not explicit. For example, in relational databases, a query $f(R_1, R_2, R_3, ..., R_n)$ can be thought of as some combination of the relations R_1,..., R_n using relational algebra operators. For simple expressions involving one binary operator, the relationship of relations R_1, R_2 to the final result is not difficult to reason about. If $f(R_1, R_2) = R_1 \cup R_2$, it is obvious that R_1 and R_2 each contain a part of the answer, although, in general, neither will contain the whole answer: R_1 and R_2 can be thought of as *consistent approximations* to the final result. If $f(R_1, R_2) = R_1 \bowtie R_2$, then every tuple in the join is contained in both R_1 and R_2 , but each relation may contain other tuples as well that do not participate in the join. Both can be thought of as *complete approximations* to the final result. Note in this case that a tuple of R_1 participating in the join with R_2 is a *partial description* of the tuple in the result since it may not contain all the fields in $R_1 \bowtie R_2$. However, for more complicated expressions like $f(R_1, R_2, R_3) = R_1 \bowtie (R_2 \cup R_3)$, it is difficult to reason about the information in R_2 and R_3 with respect to the final result.

Using the semantic notions of complete and consistent approximations, we have recently presented a method of iteratively combining structures as they become available [7,8]. That is, the user first specifies the semantic relationship of the answer to the query to the individual structures in the database. The system then combines the approximations as structures become available in such a way that a partial answer is *always* available. The partial result is represented at any point in the computation by a *bounding pair* (A, B), where A is a complete approximation of the final result and B is a consistent approximation of the final result. From the set A, the user can infer

tuples that are definitely *not* in the answer to the query; from the set B the user can infer tuples that definitely *are* part of the answer. Given two bounding pairs for a query, (A_1, B_1), (A_2, B_2), we combine them into another bounding pair (A, B) where A is no "larger" a complete approximation than A_1 or A_2, and B is "at least as large" a consistent approximation as B_1 and B_2. That is, the new bounding pair is a better approximation of the final result since it squeezes the complete and consistent approximations closer together. This continues until there are no more bounding pairs to encorporate, or until A and B describe the same set of objects, *i.e.*, the answer is completely determined. Furthermore, the partial answer can be shown to improve monotonically.

Expanding on the blood bank example, suppose that hospital X maintained a relation $NEAR-BANKS(Code, Name, Phone)$ of blood banks in its area, and that a central authority on blood banks maintained a relation $SOURCES(Code, Address, Phone...)$ (to which hospital X had access). Furthermore, assume that the required computation from the each of local blood bank databases was represented by $AMOUNT_i(Code, Quantity)$, and that the query was "Give me the *Name*, *Phone* and *Address* of all local blood banks, as well as the *Quantity* of O- on hand." Then $NEAR-BANKS$ and $SOURCES$ are each complete approximations of the final result, and $AMOUNT_1$, $AMOUNT_2$ and $AMOUNT_3$ are each consistent approximations of the final result. The initial bounding pairs would be: $(A, \{\})$ for each complete approximation A, and $(\perp, B)$ for each consistent approximation B (where $\perp$ is a special set that underdescribes any set). If, for some reason, relations $AMOUNT_2$ and $AMOUNT_3$ could not be obtained by the deadline, a partial answer of $(SOURCES \bowtie NEAR-BANKS, AMOUNT_1)$ could be constructed. That is, the complete approximation of the final result would be a relation $TEMP(Code, Address, Phone...)$, where only the local blood banks would appear. If the first blood bank could supply enough O- blood, the answer would be sufficient; however, even if it couldn't, the complete approximation would at least give the phone numbers of the remaining two blood banks so hospital X could make a phone call to determine the amount on hand. That is, the partial answer may be useful.

The system has several advantages: (1) it is not tied in to any data model in particular (although the example given was relational in flavor); (2) it detects anomalies in the database, which can arise either due to incorrect semantic understanding of the structures in the database, or due to errors contained in the database; and (3) the deadline of a query can be met by providing a partial answer. The first advantage is especially important in real-time databases since many of the structures that need to be stored do not correspond to first-normal form relations, or accepted structures in other database models (*e.g.*, system information such as stacks and queueus, and historical data). The second advantage can be demonstrated through the blood bank example: If the telephone numbers recorded in $NEAR-BANKS$ and $SOURCES$ differed, the user may want to know that they

differed rather than having the system make an *ad hoc* decision about which was correct.

A disadvantage of this approach is that the complete approximation of the query may be a very large set, and could take too long to enumerate as a partial answer. We would therefore like to be able to use rules as a shortened, but accurate, description of this set (as proposed in [9]). For example, if a monitoring system in a hospital was asked to provide the results of a routine series of tests performed on a patient ("G-series"), all of which came back with normal results, the system should avoid listing each test individually but abbreviate with "G-series normal".

References

[1] H. Wedekind and G. Zoerntlein, "Prefetching in Realtime Database Applications," in *Proc. ACM-SIGMOD International Conference on Management of Data*, pp. 215–226, 1986.

[2] December 1987. ONR Funding Workshop, San Jose, CA.

[3] E. Jenson, C. Locke, and H. Tokuda, "A Time-Driven Scheduler for Real-Time Operating Systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 112–122, December 1986.

[4] K. Lin, S. Natarajan, and J. Liu, "Imprecise Results: Utilizing Partial Computations in Real-Time Systems," in *Proceedings of the Real-Time Systems Symposium*, pp. 210–217, December 1987.

[5] D. W. Leinbaugh and M. Yamini, "Guaranteed Response Times in a Hard-Real-Time Environment," *IEEE Transactions on Software Engineering*, vol. 6, pp. 1139–1144, December 1986.

[6] R. Abbott and H. Garcia-Molina, "What is a Real-Time Database System?," in *4'th Workshop on Real-Time Software and Operating Systems*, pp. 134–138, July 1987.

[7] P. Buneman, S. Davidson, and A. Watters, "Querying Independent Databases," *Information Sciences: An International Journal*, 1988. To appear.

[8] O. Buneman, S. Davidson, and A. Watters, "A Semantics for Complex Objects and Approximate Queries: Extended Abstract," in *Proceedings of the 6'th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, March 1988.

[9] T. Imielinski, "Intelligent Query Answering in Rule Based Systems," *The Journal of Logic Programming*, vol. 4, pp. 229–257, September 1987.

PANEL

BUILDING REAL-TIME KERNELS

Chairman

Andre M. van Tilborg
Office of Naval Research

Panelists

Ashok. K. Agrawala
University of Maryland

Robert P. Cook
University of Virginia

Alan C. Shaw
University of Washington

Kang G. Shin
University of Michigan

Hideyuki Tokuda
Carnegie Mellon University

Horst F. Wedde
Wayne State University

NOTES:

NOTES: